

PIE DEVELOPERS

Volume 2.2 • March, 1994

\$8.95 US • \$9.95 Canada

Shareware, Newt Boot,
ViewFrame Reviews

Newton Memory
Management

ViewFrame Primer

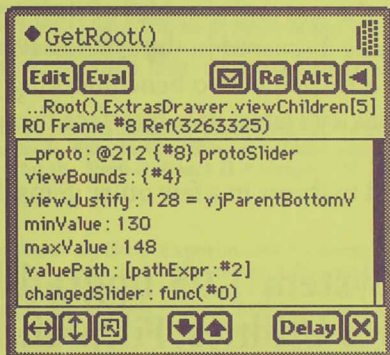
Perfect Beast II

Benchmarks

Souped-Up

Floating Calculator

The Waiters



♦View as: viewJustify
22
0x16
= vjCenterH
+ vjCenterV
+ vjParentCenterH

Size: 13 x 13



Mask:

::

Hilited:



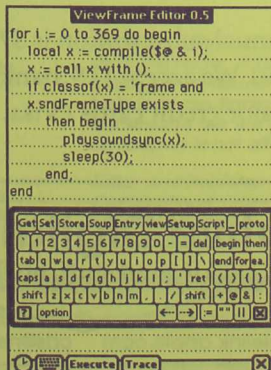
Entry in Internal: System
216 bytes (17 text)

```

tag: "UserDictionary"
data: <dictdata:#159>
_uniqueID: 3
_modtime: 47260508
    
```

```

func(unit) begin
  IF NOT(:TrackHilite(unit
    :buttonClickScript():
    :hilite(nil);
  16: true
end
    
```



```

GetRoot()
GetGlobals()
GetStores()
GetUnionSoup("")
GetView('viewFrontMost)
debug("")
/RemoveSlot(%,')

/DV(%)
/GetView(%)
    
```



The PIE DEVELOPERS Source Code Disk

It includes the text and graphics of each article in this issue, and the source code for each of the programs discussed in the articles: Perfect Beast II, Souped-Up 1.11, Floating Calc, a ViewFrame 1.0 demo, two benchmarking programs, a dimmable text button proto, and Howard Oakley's latest user interface proto, The Waiters.

We decided to throw in a few other items that we culled from various places:

the System 1.05 upgrade, CopyMachine, Disabler, The Figgles Utilities, FileMail, Graphing 101, Internals, Ladle, LlamaDOS, MailPhones, MailSaver, Memory, Peekaboo, Pour, QuickFigure Lite, ROM_Bitmaps, Rom Sounds, StewPot, Stork, Strainer, TermLimit, Tracer, TrashCan, two user group newsletters, and a whole lot more.

We didn't think we could fit all that into 1.4 MB.

Individual disks are \$15, an annual subscription is \$75. With a printed subscription the disks are only an additional \$50. We accept U.S. bank-drawn checks (made out to Creative Digital Systems), EFTs, and major credit cards (Visa, MasterCard, and American Express).

Creative Digital Systems

All good stuff, no fluff.

293 Corbett Avenue
San Francisco, CA 94114

cdssem@aol.com • cds.sem@applelink.apple.com • cds@netcom.com • 72722.3225@compuserve.com

415.621.4252

415.621.4922 (fax)

EDITORIAL

NewtonScript is Too Slow • George Henne •

NEWS, YAHOOOS AND ERRATA

Steve Mann • 3

PREVIEWS AND REVIEWS

Shareware 2.0 • Bill Colsher • 9
Newt Boot • John Marman, Steve Mann • 9
ViewFrame • Bill Colsher • 10
All About the ARM RISC Chip • Howard Oakley • 10

GETTING STARTED

Intro to Newton Memory Management • Paul Potts • 12
An Introduction to ViewFrame • Jason Harper • 14
Building the Perfect Beast II • Gregory N. Christie • 17

CHUNKS

More Amazing Soup Experiments • George Henne • 23
Of Benchmarks and Bottlenecks • Howard Oakley • 24
Spelunking With ViewFrame • Rob Bruce • 26
Subject: Startscreen Newt • Bill Kearney • 27
How to Dim a Text Button • Albéric Müller • 27

APPLICATION COOKBOOK

Souped-Up: A Soup Browser • Theo Heselmans • 28
A Different Kind of Calculator • Rob Lewis • 33

SERIOUS BUSINESS

Interface: Waiting for Newton • Howard Oakley • 32

TIPS AND TECHNIQUES

Howard Oakley • 8, 43

PIE DEVELOPERS

is copyright, published bimonthly, and trademarked by

Creative Digital Systems

293 Corbett Avenue • San Francisco, CA 94114
415.621.4252 • 415.621.4922 (fax)

cds.sem@aol.com • cds.sem@applelink.apple.com
cds@netcom.com • 72722.3225@compuserve.com

All rights reserved.

A one-year subscription to PIE Developers costs \$60 in the US and Canada, \$75 elsewhere. Newton, NewtonMail, MessagePad, NewtonScript, Newton Intelligence and the light bulb logo are trademarks of Apple Computer, Inc. All other trademarks are the property of their respective owners. PIE Developers is not affiliated with Apple Computer, Inc.

Editor and Publisher
Contributing Editor
Editorial Advisor
Technical Review

Steve Mann
Howard Oakley
Kent Sandvik
Mike Engber

PIE Developers is always interested in articles by qualified PIE programmers. Please direct inquiries to Creative Digital Systems.

NEWTONSCRIPT Is Too Slow

George W.P. Henne

The Eastwood Group

72714.3043@compuserve.com

Copyright 1994 by George P. Henne. All rights reserved.

I've been watching with interest the results of various benchmarks on the Newton, and running my own as well. There's not a lot to be enthusiastic about, it seems, and it made me give some thought as to why and what might happen.

Over the past 20 years, I've been involved with a number of languages. Some of them have had similarities to NewtonScript in that they were interpreted and developed by major computer manufacturers. With this background, I can make some guesses as to what might be happening. Remember that I don't work for Apple, nor do I have any inside information. What follows is merely my own speculation.

In the benchmarks I ran for my articles in *PIE Developers*, the speed of soup operations didn't look so bad. I was able to sustain 25 soup reads per second, a rate I would have been happy to have on minicomputers a few years ago. Another bright spot is the view system: opening and showing views appears to be adequately fast, at least in comparison to some other operations.

WAY TOO SLOW

That being said, I think the speed of NewtonScript overall is appalling. Many interpreters are on their first implementation; you don't expect them to be optimized. On the other hand, I know from years of experience that ten times improvement in interpreters is not out of the question. Can something similar be done with NewtonScript?

Interpreters have certain inherent advantages over compilers when it comes to performance tuning. The low memory overhead certainly cuts down on memory shuffling. The interpreter can be much smarter in eliminating duplicate routines by keeping needed data around, where a compiler would recompute it. There never seems to be an end to ways you can tweak an interpreter – there are so many possibilities.

Since many compilers these days reduce source code to a series of calls to built-in routines, they are not much different from interpreters, which do essentially the same thing. In many applications, both environments spend most of their time running these built-in routines. Optimization becomes a matter of speeding up getting from one routine to the next, and cutting out unnecessary calls.

On this basis, it there potential for improvement in the Newton? I think so. The fact that certain aspects – such as soup I/O and view management – look promising makes other parts of the architecture look like underperformers. The ARM chip is quick. `foreach` and assignment statements shouldn't eat it alive. Furthermore, coding these operations to run fast is not new technology.

WHAT'S THE PROBLEM?

If NewtonScript's performance is not improved, I don't think it will be due to any technical problem or the inability of the PIE development group. It will be more likely:

Optimal Use of Technical Resources. There is tremendous pressure on the Newton Development team to add new functionality and deal with new platforms. Taking people away from this activity to tune the existing product would upset the marketing folks and key customers. The PIE development team is riding a tiger – they're out in front in the PDA market, but if they fall off, they will be eaten by the competition. There is a schedule of new products, based on Newton technology, to be released by Apple and its partners over the next year. NewtonScript needs to be ready for all those products. They've got to make an either-or decision.

There is Always the New Box. The new ARM chip promises twice the speed. Why not just wait for it, and let the customers upgrade? Historically, this has been the response of hardware manufacturers when dealing with this kind of problem. New generations of hardware seem to come more quickly than new generations of software these days, and they deliver predictably better results without throwing a lot of money at uncertain software development.

Fear of Code. The parts that need to be tweaked are inevitably the ones deep in the internals, such as the evaluation routine and basic data manipulation. I know it always takes me weeks to build up the guts to make changes in these areas on products I've developed myself. I remember how tough it was to get these parts working right in the first place, the long hours and the exaltation when they finally, magically started working right. Mess with them again? No way. Ironically, the changes usually only take a few hours when I finally run out of excuses, and I wonder why I thought they were so bad to do.

RAM is Scarce. A lot of NewtonScript is burned into the ROM. Making changes involves new ROM, not a possibility with the MessagePad, or using up precious RAM space for a patch. RAM space isn't unlimited. It's likely that if a lot is needed to patch the interpreter, it will have to be stolen from the user area. I venture there's not much there that users are willing to give up, especially on Newtons without memory cards. What's there needs to go the highest bidder. Overall, the need to add code for a new feature required by a partner (such as an interface to a cellular phone PCMCIA card) will probably take precedence over a tweak to improve performance.

If it Ain't Broke, Don't Fix It. Overall, the Newton works surprisingly well for a first generation product. It's a big, rich environment that in its first release seems to have about the same number of problems as Windows (a less ambitious project) after a couple of releases. Reasons to go back and re-engineer the architecture are just not as compelling as they might be if there were serious problems. The risk that the new, improved version might introduce problems that weren't there before must be downright scary to the marketing folks.

Maybe We'll Give Them a Compiler. At the Newton Developer's Conference, the PIE development group demonstrated a compiler. Quite sensibly, it can be used on just a selected portion of code that is running too slowly, leaving the overall NewtonScript runtime environment intact. It won't speed up execution of built-in routines, but it will speed up the execution of code in between those calls. The applications they demonstrated were compute bound operations that improved from 3 to 9 times once compiled.

I think the speed of NewtonScript overall is appalling.

CAN COMPILERS REALLY SPEED THINGS UP?

The compiler didn't surprise or impress me too much. In the mid-1980s, I was a guru for a widely used version of BASIC. It was an interpreter, and the user base was clamoring for more performance. Conventional wisdom was that the interpretive nature of the product was the problem, and only a compiler would fix it. Two separate development groups got to work on compilers.

I was invited to Atlanta to benchmark the first compiler that was ready. It worked well – the execute-bound jobs I threw at it all ran three to 10 times faster. It was beginning to look like a winner. But for my final test, I threw a real, live application at it. It only ran 20% faster. Why? Well, normal applications just don't do that much computation. Most of the actual application code is setting up for system calls, performing them and putting the results somewhere. Compilers just don't help much on those operations.

Now, 20% isn't bad. But it certainly wouldn't be enough to quiet the complaints of the users. It certainly wasn't worth it in terms of what would be sacrificed in portability, run time size and convenience.

What happened? Both compiler projects were scrapped. The happy ending to the story is that a budget was obtained to go into the interpreter itself and look for improvements. By the time they were done, the product was running almost 6 times faster on the same equipment, much better than they would have done with a compiler.

THE ANSWER

I wouldn't be ungrateful to get to a NewtonScript compiler, but I would be surprised if there weren't similar opportunities for improvement as there were in that other product. Is it hopeless to look for improvements? I hope not. PIE has to see that lack of performance is crippling existing applications. It's leaving many new applications stillborn.

I spend 30% or more of my development time tuning. I wouldn't need to do that if they fixed the speed. Multiply that 30% by the 4000 developers out there and there could be a significant impact on the future of the Newton.

The above excuses are more marketing than technical roadblocks. I'm worried that the great things accomplished by Newton developers could fizzle because the Marketing Department (AKA the Sales Prevention Department) reassigns the priorities so that improvements never happen.

There is one very real hope. It seems quite possible that large portions of NewtonScript are actually written in NewtonScript. Take a look at the whole internal structure of how everything is set up in the MessagePad. It's optimized for use with NewtonScript. If a new function needs to be added, do you think they code in C++ right away, or perhaps code it first in NewtonScript until they get it running just right? That would account for the speed.

What if the primary purpose of their compiler isn't for use by developers, but to optimize the code that is running inside each MessagePad? It make a lot of sense. The tool they demonstrated at the Developer's Conference seemed to work, at least under lab conditions. That's probably good enough already for internal use by PIE DTS.

With these difficulties and potentials, we should encourage Apple to continue to work on performance. We'll all benefit, especially Apple. ☺

George Henne is with The Eastwood Group, 77 Hill Crescent, Toronto, Canada M1M 1J3. They specialize in the application of new technologies. You can reach him at 416.264.5777, 416.264.5888 (fax) or 72714.3043@compuserve.com.

NEWS, YAHOO'S AND ERRATA

Steve Mann

PIE DEVELOPERS ANNOUNCEMENTS

Starting with issue 2.2, *PIE Developers* is now available in selected technical bookstores and newsstands around the country. Although our distributor could not give us a complete list of locations by press time, look for us at Barnes & Nobles Superstores, Computer Literacy, and other retail locations that stock computer-related magazines.

We would also like to announce the *PIE Developers* Internet mailing list. Most of you have already been contacted about this via email. This is a broadcast list that we can use to send messages out to all our subscribers. We've set it up so that we can reach you quickly and effectively in between issues.

We expect to use this list to distribute selected articles from each issue, prior to the availability of the printed issue, to announce special hardware and software offers just for our subscribers, and to occasionally poll our readers. We're also toying with the idea of publishing interim electronic issues every other month in between the printed issues. If you have ideas for ways we can use this list to benefit our subscribers, please let us know.

For those that haven't been contacted about this list, we simply did not have a valid email address for you in our records. If you would like to be included, send us an email at cds@netcom.com. Conversely, if any of you want to be removed from this list, send us a message as well.

GNUT TOP 10 SHAREWARE LIST (#1 JAN '94)

The following list was compiled based on input from members of the NEWTON forum on CIS and a review by GNUT members. Thanks to everyone who participated - and especially to the winners who are helping to make our NEWTON's more useful!

The list is in order of 1st (best) to 10th. The program description has been taken from the CIS library with some minor editing.

1) List-It (LISTIT.SIT 75K)

Manages multiple TO-DO lists. Version 2.2 of the popular package for the Newton is essentially a maintenance release. It fixes a few bugs and now closes the Extras drawer automatically. By Macapa Software.

2) Keyboard (TRMKYB.SIT 29K)

Erica Liebman Sadun's keyboard is superb. Popup menus, 5 displays (alpha, num, punc, unix and ?), cursors, bulleting, <- deletes->, tab & more. Palette shrinks to a wisp.

3) CopyMachine (COPYMA.7K)

CopyMachine was written by Detlef Beyer from Cologne/Germany (BEYER1@applelink.apple.com). It's a great little program that lets you view the number of stored items on your card and in internal memory. You can then move all or selected items from or to the card with one tab of a button. Great for items backup or if you switch from a US MessagePad to the German version with keeping your entries.

4) StewPot (STEWPO.SIT 25K)

Allows the user to view and manipulate "soups" on the NEWTON. New version fixes some minor problems and adds Copy/Move of entire soups. Please delete old versions after downloading, and make sure and back up your data before attempting to make changes :-). Copyright 1993 by Mark Underwood, Maui Software.

5) Resto 1.3 (RESTO.BIN 35K)

Little application that helps you split the bill at the restaurant. This version (among other things) keeps some parameters when you close it! Three tax levels are available.

6) Date Util (DATEUT.SIT 10K)

View all of your Dates, Meetings, To do items, etc. Move and Delete items from your datebook. Version 0.91 fixes some exceptions when trying to display empty To Do items and when switching lists. Date Utility is "Newton Freeware".

7) Solo (SOLO12.SIT 50K)

This is a newer version of Renaud Boisjoly's Newton Solitaire package. It includes a new surprise ending to the game, improved graphics, and also a smaller lite version.

8) Newtris (NEWTRI.BIN 29K)

This is Tetris for the Newton. Version 1.3 includes a pause button that minimizes the screen, and now has sound. By Kendall Redburn.

9) Convert Metric (CNVRT.SIT 24K)

Convert! provides conversion tables for your Newton. The calculated measurements are area, length, volume, mass and temperature. This utility calculates more types of measurements (e.g., slugs and stones) than the Newton's built-in Metric Conversion tool.

10) Hot Buttons (HOTBUT.SIT 14K)

Hot Buttons 0.9 by Cesar Maiorino. Floating palette of configurable hot buttons.

GNUT is a Toronto-based group of business users of the NEWTON. If you would like more information about the group please contact John Marman at 70410.1257@compuserve.com. The minutes of their latest meeting can be found on the PIE Developers 2.2 source code disk.

GERMAN NEWTON MESSAGEPAD

In December, 1993, Apple announced availability of the German version of the Newton MessagePad at retailers in Germany, Austria and Switzerland. The German version of the MessagePad is the first localized Newton PDA to ship in a non-English speaking area. The flexible design of the German version of the MessagePad enables the handwriting recognition feature and the integrated dictionary to be set specifically for Germany, Austria and Switzerland.

NEWTON PAGING LEASE

Apple Computer and MobileComm are now offering leases on Newton MessagePads preconfigured for mobile paging. There are three different leases available: local, citilink, and nationwide. The base 24-month lease rates are \$49.95, \$69.95, and \$81.95 respectively. Each includes a MessagePad, Apple's wireless paging PCMCIA card, and a level of paging service from MobileComm:

- Local service is for one region. It includes 100 80-character messages a month. Additional messages are \$.25 each.
- Citilink is for a region with user-controlled forwarding to another metropolitan area. Designed for the occasional traveler, citilink service includes 125 45-character messages a month. Additional messages are \$.60 each.
- Nationwide service covers all 50 states. It also includes 125 45-character messages per month, with a \$.60 surcharge over that limit.

In addition, if you use MobileComm's operators to page a MessagePad owner, the recipient is charged \$.70 per call. The leaseholder can purchase optional paging software for \$14.95. Designed to work with both PCs and Macs, this software can be freely distributed to anyone. All messages sent using the software are free to the recipient. Call 1.800.474.MESG for more information.

NEWTON TECHNOLOGY AWARDS

Newton Intelligence technology was selected for *BYTE* magazine's highest award, the 1993 *BYTE* "Award of Excellence." The Newton Intelligence operating system was commended for its ground-breaking advances in mobile computing and for being a strong foundation on which to continue advances in communications and computing.

The Newton MessagePad was also selected by *Fortune Magazine* as one of the "Products of the Year." The award honors products introduced in 1993 that have made a significant impact in their respective markets. The MessagePad received several other awards in 1993, including:

- Editors' Choice Award for "Most Promising Portable" by *PC LapTop Computers Magazine*,
- first place in *PC Magazine*'s "Design Category" of their 10th Annual Technical Excellence Awards, and
- *Reseller Management magazine*'s "Best-To-Sell Products of the Year" and "Most Innovative Product of the Year" in their 1993 Readers' Choice Awards.

The Newton MessagePad also won two awards in Germany, the "1993 Award of the Most Promising Product and Technology" by *MACWELT* and the "1993 Product of the Year" award by *DM Magazine*.

Finally, in the March issue of *MacUser* magazine, the editors dubbed Newton Intelligence the "Breakthrough Technology of the Year" because of the handwriting technology, the "bold, new approach" to data storage, the Intelligent Assistant, and the extensive communications architecture.

INFRARED NEWS

According to a message on UseNet, Apple is working on a developer kit for creating Newton applications that have custom infrared capabilities primarily for setting up a Newton as a remote controller. Developers who are interested in actually creating those types of applications and are willing to send feedback to Apple about the kit, should send a message to Kent Sandvik at PIE DTS (sandvik@newton.apple.com). The message should have the word 'REMOTE' somewhere in the subject line. (By the time you read this the offer may have expired).

In a somewhat unrelated item, anyone interested in getting details of the Newton's Sharp-designed infrared hardware and low-level protocols should contact Robert Stuart at Sharp Electronics. He is exceedingly helpful and is able to send interested developers application notes, IC specifications, and protocol details. It's not the whole puzzle but it's some of the pieces. Robert Stuart, Sharp Electronics Corporation, 5700 NE Pacific Rim Boulevard, MS 20, Camas, WA 98607-9489. 206-834-8948, 206-834-8903 (fax), stuart@secms.sharpwa.com.

STARCORE (KINDA) RUNS AMUCK

In January, in yet another round of fairly vaporous product announcements, StarCore announced 12 new titles for the first quarter 1994. Seven of these titles will be published, five will be distributed as affiliate-label products. In the fourth quarter of 1993, StarCore published six titles and distributed another four affiliate-label packages.

The scheduled new published releases are:

- *Money Magazine* Financial Assistant which helps the user calculate financial decisions including mortgages, refinancing, bond prices/yields, budgeting, buying versus leasing and percentages and statistics (\$69.95);
- *Economist World in Figures*, detailed facts and figures for more than 60 countries, covering national economies, populations, family life, geography and trade (\$99.95);
- *Solitaire and Other Card Games* - Solitaire, casino-style Black Jack, Baccarat and Poker (\$39.95);
- *Jig Saw Puzzle Strategy Game*, a one- or two-person puzzle game (\$39.95);
- *Time Out*, a London travel guide that features detailed information on what to do, where to stay, museums and art galleries, restaurants, shops transportation and entertainment (\$119.95);
- *DrawPad*, targeted primarily for the architect or designer, a tool for field sketching and estimating do-it-yourself and "back of the envelope" calculations (\$69.95);
- *Dyno NotePad*, an outlining tool (\$59.95);

In addition, StarCore plans to distribute:

- *TaxPro*, tax planning software for the Newton (\$49.95);
- *Personal Time & Billing* (\$149);
- *Silicon Casino* - video poker, blackjack, craps, slots and baccarat (\$59.95); and
- *PresenterPad*, a suite of tools for creating and managing effective slide presentations and speeches - and turns the user's Newton device into a portable prompter, slide note manager or remote slide controller (\$89);
- *PocketCall*, which provides access to on-line communications services, including CompuServe and America On-Line (\$149);

Now if StarCore can just get these titles out the door and on the retailers shelves, this batch of killer apps might rekindle the MessagePad's lagging sales.

In a separate announcement, Avail Technology Inc. announced GeoAssist and a StarCore distribution agreement. GeoAssist, with a suggested retail price of \$59, includes a database of time-zone maps, major cities, NewtonMail access numbers, telephone calling codes and reference numbers, air travel information and more. Designed for serious mobile professionals, GeoAssist includes a FlightTrack™ option which continuously tracks an airplane flight in progress. Now *that* should get Newtons flying off the retail shelves.

PCMCIA NEWS

During the first quarter of 1994, several manufacturers announced new PCMCIA memory products for the Newton MessagePad. MicroData Group, Inc. of Boxford, Massachusetts announced Intel-brand PCMCIA Flash RAM cards. At the announcement, the two MByte card was priced at \$259, the four MByte card at \$399. MicroData Group can be reached at (508) 922-1821 or 72156435@compuserve.com.

Lifetime Memory Products, a developer specializing in the design, manufacture and marketing of memory boards for all personal computers, announced the availability of 4 MByte Flash cards in production quantities. In recognition of the volatility of the PCMCIA memory market, Lifetime did not publish prices. Contact them for current pricing.

Lifetime also announced that they now manufacture PCMCIA ROM cards in capacities from 256K to 20 MByte per card, in both OTP and Mask-ROM versions. They can be reached at (714) 969-2421 or LIFETIME@applelink.apple.com.

In a related story reported in the January 31 issue of *MacWeek*, Apple and Intel have forged an OEM agreement that

guarantees Apple the lowest possible prices for Intel's 2 MByte Flash RAM cards. In exchange for purchasing the bulk of Intel's Flash RAM production, Intel apparently agreed to guarantee Apple the lowest prices on Intel-based PCMCIA RAM cards, agreed not to compete with Apple, and also guaranteed that third-party Flash RAM cards using Intel chips would not be Newton-compatible.

It's possible that a similar arrangement might emerge for four MByte Intel Flash RAM cards at some time in the future.

LLAMA TOSSING

From: sandvik@newton.apple.com (Kent Sandvik)

Newsgroups: comp.sys.newton.announce

Subject: Serious Newton Terminology Discussions at PIE

Date: Thu, 20 Jan 94 07:58:29 PST

Was reading one of our internal groupware documents where we discuss Newton terminology. Just to provide a sneak view of the serious work we do inside our group, here's an entry about one of the terms we argue about currently:

Throw a Llama

A classical situation where the Newton Programmer is getting a sudden insight into how the objects are referred to in memory.

- cjb: Such insight is usually attained after months of solitary meditation in front of the Inspector window, but occasionally it can be transmitted immediately by a master to a promising student.
- KS: I'm still trying to obtain this sudden insight, Zen is far simpler than NewtonScript internals.
- ba: but does a Newton Programmer have Buddha-nature?
- gk: Mu! (Moo?! Woof!?) [Does a DogCow have Buddha-nature?]
- KS: What is the sound of a llama spitting on itself?
- ds: Llamas, like sleepy meditators, only manage to drool. They never really achieve true spit. But do Newton Programmers and dogs and cows NEED Buddha-nature?
- gk: Moof! (Is what I should have said.)
- KS: When you see a llama picture in a Newton manual, rip out the page.

WILLSTEIN SCIENTIFIC SOFTWARE SHIPS BARCODE SOFTWARE

In February, WillStein Scientific Software (WSS) Incorporated began shipping BarCoded® for Newton PDAs. BarCoded is software that automates the collection of barcode information. It controls various input devices, collects and stores data, allows user handwritten inputs, and uploads the information to selected computer systems. BarCoded stores the input in a user-defined folder; the records are displayed in a table format and numbered as they are collected. As each scan is completed, the user can include a handwritten comment or enter an amount with units.

The collected information can be transferred by docking the Newton to a Macintosh, Windows, DOS or mainframe machine with a serial or modem connection. The default file format is tab-delimited, but the program can be set up for any delimiter or fixed field lengths.

BarCoded is distributed on both PCMCIA cards and floppy disks (Macintosh and DOS). The disk version has a manufacturer's suggested retail price of \$299; the PCMCIA version is priced at \$499. BarCoded requires communications software on the desktop for data transfer. It also requires an HP barcode wand and cable, a PSC 5312 laser gun and cable, or a user-supplied scanner.

WillStein Scientific Software Incorporated
1723 Elmwood Ave., Wilmette, IL 60091
708.256.2895, 708.256.5220 (fax).
willstein@applelink.apple.com

NOTABLE QUOTABLES

"The Newton will continue to flounder until Apple either lowers its price, adds decent handwriting recognition, or at the very least, adds cool handwriting editing technology like what's available from ex-Apple guy Greg Stikeleather's company, Aha. Gina Smith, *San Francisco Examiner*.

"We will never have perfect handwriting recognition. People can read handwriting reasonably well [only] when they know what it is about. Pens do have a place for form filling, short writing, drawing, and selection. Donald Norman, Apple fellow and author of *Things that Make Us Smart*, *MacWorld* magazine.

"It's time to build a better Newton. The MessagePad I bought with such hope and excitement [has] proven almost useless to me. Give this "communications assistant" some communications capabilities [like a] built in a high-speed, full-featured fax modem. Newton's default recognition system is dictionary-based. [The problem is that] the MessagePad is marketed as a way to manage the "jumble of details" that make up our lives, [but] few of the details that make up my life are dictionary terms. Henry Norr, *MacWeek*.

"The much-heralded age of personal digital communicating has gotten off to a rocky start. But just wait. The current problem can be summed up in one word: Newton. Apple Computer Inc. raised expectations by heavily promoting its Newton MessagePad as the first personal digital assistant... When the product arrived in stores, it couldn't recognize handwriting particularly well, couldn't communicate at all without lots of costly additional gear and was widely ridiculed. The Newton, says Microsoft Corp. Chairman William H. Gates, may have 'set the category back a couple of years.' G. Christian Hill and Jim Carlton, *The Wall Street Journal*

WALL STREET JOURNAL SLAMS PDAS

In a series of news articles about PDAs published over the last several weeks, the Wall Street Journal has painted a less-than-flattering portrait of the market. Although Apple's Newton MessagePad has been a prime target, other PDAs, including some not even released, have been mentioned. Here are a few sample headlines:

"Apple's Sales Data Suggest to Analysts That New MessagePad is Floundering.

"Compaq to Delay Hand-Held Computer After Bad Reviews of Pen-Based Rival"

"First Hand-Held Data Communicators Are Losers, but Makers Won't Give Up"

"AT&T's Eo to Focus on Cellular Phones As Market for Data Devices Proves Slow"

"The Newton has given personal digital assistants a bad name."

Short on facts, but long on "analyst" observations, one of the articles actually says that the PDA market is in its gestation and that it will take a few more product generations before the concept really takes off. Another suggested that PDAs will in fact revolutionize personal communications... eventually. Now if they could only report *those items* in 48-point headlines, instead of early predictions of the PDA's demise. That would be news.

NEWTON SYSTEM 1.05 SHIPS

In late January Apple announced Newton system update version 1.05. System update 1.05 improves memory and power management and optimizes system performance for third party applications. It gives users the same functionality as provided by system 1.11. The system update uses an additional 32K of system memory to make a more reliable user experience.

To receive the update at no cost, Newton MessagePad owners in the United States need to contact the Newton Fulfillment Center at 1-800-242-3374. Outside the United States, contact your local Apple Authorized Service Provider.

In the United States, there are three ways to update to system 1.05:

- on a floppy disk;
- on a memory card;
- from various on-line services such as AppleLink, CompuServe, America Online and others.

In order to install system update 1.05, customers must first back up their Newton data on a PCMCIA card or on their desktop computer using Newton Connection Kit.

LLAMADOS: A POST-NEWTON COMMAND LINE INTERPRETER

Reprinted from a press release found floating on the ether.

While the Newton is a tremendous design, it lacks a feature found on most computers: a command line interface for system management. Rather than invent a new interface for the Newton, Llamados implements the world's most widely used interface, with well over 40 million users: DOS. The developers at RegSoft have done their best to make sure that the quality and user friendliness of the DOS interface survived intact on the Newton in their new product, LlamaDOS 1.06.

With LlamaDOS, a user can traverse directories, look at files and the status of their system using the same commands they have used for years on PCs. Of course, not all commands are implemented: just a subset that will be used regularly. Any application can also be run directly from the command line.

"It's really increased my productivity," said Frank Schaedlich of HHI Inc, one of the beta testers. "I also feel I have much better control over the internals of the Newton. It was great not having to learn a new interface."

The normal cost of the software is \$195.00 US. However, a special limited time competitive upgrade of just \$19.95 is being offered. To qualify, please specify the Newton OS you are upgrading from. Revenues will be used to produce enhanced versions of LlamaDOS. Look for the shareware copy on the networks, or order directly from Don Vollum, RegSoft, PO Box 1894, Portland, OR 97207. For more information contact George Henne, RegSoft, 72714.3043@compuserve.com

LlamaDOS is named after the cuddly, doe-eyed mascot of the Newton team. The use of RegSoft as a company name by no means implies that RegSoft or its employees or agents are in any way taking a position in the ongoing Ned verses Reg dispute.

LlamaDOS can be found on the PIE Developers 2.2 disk.

QUICKFIGURE LITE

From: moana@teleport.com (Charles Vollum)

Newsgroups: comp.sys.newton.announce

Subject: QuickFigure Spreadsheet available

Date: Sun, 30 Jan 1994 20:48:59 -0500

A new spreadsheet app for Newton called QuickFigure Lite is available on the newton.uiowa.edu FTP site in the file pub/newton/software/apps/quick-figure.hqx. QuickFigure Lite is a fully functional spreadsheet for the Newton. It supports all standard Newton math and financial functions, allows the user to store multiple worksheets, has printing and faxing capability, and many other features. QuickFigure Lite is shareware. Registered users receive a manual and an enhanced version of the software.

QuickFigure Lite can be found on the PIE Developers 2.2 disk.

SELF 3.0

Reprinted from comp.lang.smalltalk.

The Self Group at Sun Microsystems Laboratories, Inc. and Stanford University is pleased to announce Release 3.0 of the experimental object-oriented programming language Self. [NewtonScript is based on Self. Ed]. This release provides simple installation, and starts up with an interactive, animated tutorial. Self currently runs on SPARC-based Sun workstations running SunOS 4.1.x or Solaris 2.3.

Designed for expressive power and malleability, Self combines a pure, prototype-based object model with uniform access to state and behavior. Unlike other languages, Self allows objects to inherit state and to change their patterns of inheritance dynamically. Self's customizing compiler can generate very efficient code compared to other dynamically-typed object-oriented languages.

The latest release is more mature than earlier releases: more Self code has been written, debugging is easier, multiprocessing is more robust, and more has been added to the experimental graphical user interface which can now be used to develop code. There is now a mechanism (still under development) for saving objects in modules, and a source-level profiler.

The Self system is the result of an ongoing research project and therefore is an experimental system. We believe, however, that the system is stable enough to be used by a larger community, giving people outside of the project a chance to explore Self.

This release is available free of charge and can be obtained via anonymous FTP from Self.stanford.edu. Also available for FTP are a number of published papers about Self. There is a mail group for those interested in random ramblings about Self, Self-interest@Self.stanford.edu. Send mail to self-request@self.stanford.edu to be added to it.

>>> PCMCIA NOW!! <<<

Author: STEALTH@applelink.apple.com

I know Apple wants apps being developed without them being dependent on the actual hardware design. But what I really need for a project that I am about to undertake is ...

PCMCIA event-triggered Newton Script calls.

We do not need to know the internals of Newton but we VERY much need to get to call PCMCIA card and socket services via Newton scripts. They could be scripts like...

```
viewCardInsertScript()  
viewCardRemoveScript()  
viewCardSpecificScript() // for card triggered events  
viewCardPowerChangeScript()  
viewCardServiceConfigChangeScript()
```

and other things like that.

If this doesn't happen soon. There will be a BIG gap in available cards that could work with other PDA and not Newton.

Remember that there are _hundreds_ of little companies out there coming up with PCMCIA cards that are already or will be soon with both WinPad and Zoomer. I very much love the Newton and I shifted my career to do development for it. I very much want it to win. But remember the Macintosh made it 'cause of very little companies with almost no money but a big faith.

This will happen again. Please Apple PIE, don't shoot yourself in the foot.

INTERNET NEWS

Comp.binaries.newton

There is a new Usenet news group for posting Newton applications in both binary and source code form. Free software, shareware, and demoware are all suitable for posting (demoware is demo versions of commercial applications which are artificially limited in functionality in some way). Software might include applications, Newton Books, or other Newton-related non-textual material which, in the opinion of the moderator, is of general interest to the Newton community. This news group will not include product announcements or items of a strictly promotional nature, but it might include upgrades, patches, and the like. The moderator will also maintain FTP and gopher archives of all postings.

Newton FTP Archive on the Move

As of January 12, 1994 the Newton archive site at Johns Hopkins University has been replaced by newton.uiowa.edu, located at the University of Iowa. The directory structure is similar to the JHU site, with a few additions.

Robert Bruce, moderator of the JHU site, is assisting David Rarick in maintaining the new site. New files may be uploaded to /submissions. These files will be placed in /pub/newton. Send questions, comments, or suggestions to Ric Mommer at ric-mommer@uiowa.edu or David at david-rarick@uiowa.edu.

Newton Archive Mirror Site

Dirk Husemann of the University of Erlangen-Nuremberg is in the process of installing a mirror of the newton.uiowa.edu FTP archive at ftp.uni-erlangen.de/pub/Newton. Dirk can be reached at Dirk.Husemann@informatik.uni-erlangen.de

Newton FTP Site

The Manitoba Macintosh Developers Association is pleased to announce that they have made a portion of their Macintosh FTP site available for Newton users. This site is located at FTP ftp.cc.umanitoba.ca in the /Mac-Develop/Newton directory. For more information contact David A. denBoer at the University of Manitoba (denboer@CC.UManitoba.CA).

PCMCIA FTP site

There is a collection of PCMCIA information collecting on an anonymous FTP site at csd4.csd.uwm.edu in the directory /pub/Portable. Gathered by Anthony Stieber from a variety of predominantly Usenet sources, the documents are a good starting point for collecting PCMCIA hardware information.

Another place to look for PCMCIA information is at PCMCIA, the official organization that carries the torch of PC-card standardization. They can be reached at PCMCIA, 1030 East Duane Ave., Suite G, Sunnyvale, CA, 408.720.0107, 408.720.9416 (fax). They have a variety of documents and membership plans available for a fee.

PEEKABOO: THE PROGRAMMER'S FRIEND

Peekaboo, a new shareware program from prolific Newton developer Erica Sadun, lets you turn Inspector tracing on and off from the Newton while you debug. Peek through your software as it runs, or peek through global system calls while you experiment. Additional buttons let you set detail depth and call up the "Inspector" and "Toolkit" applications. All you have to do is connect up the Inspector to NTK. Peekaboo is priced at \$12.50.

Peekaboo is included on the PIE Developers 2.2 source disk.

BEAR RIVER ASSOCIATES ENDORSES NEWTON

Bear River Associates announced in Berkeley, California that it is now providing Apple-qualified Newton programmer training. The five-day course teaches programmers how to develop Newton applications using NewtonScript and the Newton Toolkit. Courses primarily take place at customers' sites, however public courses will be offered on an occasional basis. Bear River also provides Newton consulting and development services.

"I am very enthusiastic about Newton and see mobile computing as the wave of the future," said Anthony Meadow, President of Bear River Associates. "The Newton will enable many types of applications which are not possible even with laptops such as PowerBooks. Enterprises are finding that mobile computing applications can give them a competitive advantage, especially for some of the early adopters."

Bear River began working with Apple before Newton's release. They try to make sure that all their programming instructors are working software engineers. Their Newton instructors began developing their first applications as soon as they received their first Newton.

Bear River Associates provides a complete line of services for Macintosh, Windows, Newton, and other business desktop environments, including programming, systems integration, training, and consulting.

PIE DTS BEEFS UP TECH SUPPORT

In early February, PIE DTS engineer Mike Engber became the father of a new baby boy: seven pounds, 16 ounces, Colter Avraham Engber. Congratulations Mike. How soon can we get your son out on the nets to answer some questions?

PIE DEVELOPERS 2.1 ERRATA (COURTESY PIE DTS)

Note: Although we listed Kent Sandvik and Mike Engber of PIE DTS as editorial advisor and technical reviewer (respectively) in issue 2.1, in fact we did not show them the complete contents of that issue for review. As a result, our technical accuracy was not what it should have been. The fault is entirely ours and is in no way a reflection on their prodigious talents. This time they previewed almost everything for issue 2.2.

TIPS AND TECHNIQUES

by Howard Oakley

TRACKING ENTRIES IN SOUPS

If you want to keep track of soup entries, you should not rely on the `_uniqueID` field, which is only guaranteed to be unique within a single soup. A more reliable technique is to keep an array of unique IDs, which you assign yourself as a field within each entry. If you then use the array to search for IDs, the index allows you to do a `:Move(index)` to locate the actual entry in the soup.

NEAT INSPECTOR FUNCTIONS

Try this script in the Inspector window, to get a good overall view of functions:

```
neatinterp := func()
begin
  printDepth := 1;
  local arr := foreach slot,y in functions collect
    slot & " args: " & if y.numArgs then
      y.numArgs else 0;
  Sort(arr, '|Str<|,nil);
  foreach element in arr do Print(element);
end;
```

Making the top level view accessible (page 17)

Your application's base view is already accessible via its application symbol – there is no need to do anything special, just use your app symbol to refer to your base view. Using `declareSelf` as suggested in the tip is not only unnecessary, it will cause your application's close box not to work.

There's a Fly in My Soup (page 29)

There is a line of code that reads:

```
theCurrent := theCursor.current;
```

`current` is an undocumented and unsupported slot in cursor. If you need the entry a cursor is pointing to, use the `Entry` message:

```
theCurrent := theCursor:Entry();
```

Also, the first section of the article, Find the Culprits, is unnecessary, inefficient, and unsupported. There is a documented global variable called `userConfiguration` that is the entry of interest. Simply use this global in your code (for instance, `userConfiguration.userFolders`).

Conference Report From Michigan - Day Two (page 7)

The session on Soups & Stores was given by Mike Engber. The session on System Services was given by Maurice Sharp.

PIE DEVELOPERS 2.1 ERRATA (COURTESY OF OUR READERS)

Table of Contents (page 1)

Howard Oakley's name was incorrectly spelled Howard Oaky. Sorry Howard.

Building the perfect Beast (page 18)

In the second column, our Postscript level 2 printer driver dropped the project character (option "p") in several locations, most notably in project names and other essential places where you would expect to find it. We would show you which character that is in this correction, but we're still trying to figure out how to get it to print. Stay tuned... 🐾

Another good Inspector call is:

```
GuessAddressee("yourName");
```

which returns a list of name slots containing "yourName" as the starting string in any field.

INFRA-RED ASYMMETRY

Communications using the IR port have to be asymmetric – at any time, one port has to be receiving, the other transmitting. If you wish to try novel things with the port, you need to implement a protocol that ensures this.

GETTING A VIEW OFFSET

Call `:GlobalBox()` to get the offset of a view. It returns a rectangle with top and left slots containing the offset.

NEWTONSCRIPT IN A BOOK

To implement NewtonScript over the top of a page of a Book Maker book, use this code:

```
.story
<put the page contents here>
.script
// put your NewtonScript code here
.endscript
```


PREVIEWS AND REVIEWS

SHAREWARE 2.0

Bill Colsher
CreamNEggs@aol.com

Since last issue of *PIE Developers* the Newton shareware category has really taken off. A lot of new packages have hit the net and authors of existing packages are improving their products constantly. One of the most popular categories seems to be soup. This issue we take a look at three soup-related packages: SoupedUp, StewPot and the updated Strainer 0.8. Then we have Memory, a nifty memory usage monitor and even though it's not really developer shareware, the amazingly cool Graphing 101. It's one of those simple ideas that could easily develop into an indispensable tool for anyone who has trouble (like me) visualizing graphs from the high school Algebra level on up.

SOUPEDUP

SoupedUp is a straightforward soup browser from Theo Heselmans. It has a simple interface that emphasizes the information in the soup being browsed. Since it is strictly a browser, there is no possibility of accidentally damaging anything. In addition to producing one of the first complete soup browsers Theo has very kindly included the source project for SoupedUp. Including source code for utilities like this is extremely helpful, even to experienced developers. Theo deserves a vote of thanks and a hearty well done for a neat application as well as a generous nature!

[See Theo's article in this issue for a description of the latest version of SoupedUp, version 1.11. Ed.]

STEW POT

StewPot, on the other hand, is a very powerful (and potentially dangerous) soup utility. This freeware package is nothing less than a (nearly) complete soup editor/manipulator. In addition to displaying the structure and content of a selected soup, StewPot also allows you to copy or move a soup to another store, create a new soup, or remove an entire soup.

StewPot also provides a number of operations on soup entries. You can make a copy of the currently displayed entry, move the current entry to another store, remove an entry entirely, and remove all entries. StewPot doesn't stop there however. It also allows you to add and remove slots in soup entries, and to edit the contents of individual unary slots (slots that do not contain a frame or an array). The documentation included with StewPot indicates that this last limitation is being worked on and that eventually StewPot will be able to edit inside frames and arrays that are slots in soup entries.

STRAINER

Strainer (described in the last issue of *PIE Developers*) from Paul Potts has been updated with support for straining the ToDo list, the ability to mark entries in multiple soups and strain "all at once," and the ability to work with external stores. Strainer is a handy utility for almost everyone - Apple probably should have included something like it as standard equipment.

MEMORY

Memory is a tiny utility that simply displays the amount of memory in use and available on the internal store in a draggable floating view. It also has a button for garbage collection. Although Memory doesn't do a lot, it can be handy for those awkward times during development when Newton doesn't "have

enough memory to..." It can also be used to convince die hard C++ programmers that Newton really does do a pretty good job of garbage collection all by itself. (Also Memory has a really cool icon in the shape of a brain).

GRAPHING

Finally we come to a package I would have killed for in eighth grade Algebra - Graphing 101. This wonderful program provides a calculator-like keypad at the bottom of the screen that is used to enter the function to be graphed. A small area in the center of the screen displays the function as you key it. This area also accepts written input. The graphing area at the top of the screen requires only a tap to recalculate the graph and display it.

As it stands, Graphing 101 is a successful proof of concept package. It lacks a few features that would make it eminently practical (like the ability to label the axes and print the result). I hope Prescience follows up with a more sophisticated commercial version of Graphing 101. It could make the Newton virtually required equipment for anyone going beyond basic Algebra in high school or college. ☺

StewPot

Author: Mark Underwood
Status: Freeware
Address: mau@sun1.ema.com

SoupedUp

Author: Theo Heselmans
Status: Freeware
Address: Versa.Versa@applelink.apple.com

Strainer 0.8 - Update

Author: Paul R. Potts
Status: Freeware
Address: 71561.3362@compuserve.com

Memory

Author: John Calhoun
Status: Freeware
Address: rumor has it John works at Casady & Greene

Graphing 101

Author: Prescience Corporation
Status: Freeware
Address: D0588@applelink.apple.com

All of these programs can be found on the *PIE Developers 2.2* source code disk.

NEWT BOOT

John Marman
GNUT President

The NEWT BOOT is a fabric carrying case for the NEWTON. It has two main zippered compartments in which to hold the NEWTON and additional accessories. Slots for two PCMCIA cards are available. A shoulder strap can be attached to the case for carrying.

I used the NEWT BOOT at the office for a couple weeks to carry around my NEWT, the connection cable, disks and the odd snack. Although the donated case was green - not my first choice of colors - I didn't hesitate to take the NEWT BOOT into management meetings. It travels well since it can carry additional accessories - and best of all, I liked the protection the padded case provided. Like the standard NEWTON case, the NEWT BOOT has a hard protective cover to shield the screen when the case is closed.

The only criticism I have (other than the Green color – other colors are available) is that, unlike Apple's leather case, the NEWT BOOT does not use the clasp on the NEWTON's back to securely attach to the case. You have to be very careful to ensure that the case is zipped up. Note: designer Colleen Thompson added a couple of colored ties to the zipper to aid in visually checking that the unit is zipped up – very clever.

"My" NEWT BOOT was reluctantly given up at the meeting to Ray Wong from Xerox Canada, who won the door prize. Congratulations Ray!

A SECOND OPINION

Steve Mann

The Newt Boot is made of the same stuff that most hiking and backpacking equipment seems to be made of these days – a heavy-duty, nylon-based, canvas-like material called Cordura. It's sturdy, with big, heavily-stitched seams. I can't imagine this thing breaking under any circumstances.

It has two equal-sized pouches, one for your Newton, the other for accessories. The accessory compartment includes holders for a couple of pens, pencils, or styli, and two pockets for either PCMCIA or old-fashioned printed business cards. The two compartments, which can be opened and closed independently, are divided by a covered plate that seems to be made of metal. The plate is designed to protect your Newton's faceplate.

After carrying a Newt Boot around MacWorld for a few days, I have only one complaint: the big, heavy zippers have large free-swinging handles that have a tendency to clank together all the time. As a former musician and psychoacoustician (with what I've been told is an abnormal sensitivity to sounds) with the Newt Boot I feel like I would fit right in on a Chilean mountain-side with a herd of clanker-adorned llamas (maybe that's not such a bad thing). *[At press time, we learned that the next revision (1.02) will have custom neoprene zipper pulls that should be much quieter. How's that for customer service?]*

The Newt Boot beats the hell out of the cheap nylon carrying case that Apple handed out at the Newton Developer's Conference. It doesn't look as good (or as expensive) as the leather Newton case, but I bet it's a lot more practical. 

The Newt Boot is available for \$49 plus shipping and handling at 307.733.0906, 307.739.1716 (fax), 72377.2740@compuserve.com, or Newt Boot, Box 1613, Jackson, WY 83001.

VIEWFRAME

William L. Colsher
creamneggs@aol.com

ViewFrame by Jason Harper is the first major developer tool for the MessagePad to be published by a third party (in this case, Creative Digital Systems, the publisher of this magazine as well). ViewFrame consists of three applications:

- ViewFrame - a utility for examining and editing NewtonScript objects
- ViewFrame Editor - a tool for writing simple programs on the MessagePad
- Programmer's Keyboard - to ease on screen entry of NewtonScript

The full ViewFrame package requires a bit over 100K. Given the recently released 1.05 system, you need a storage card. You'll also want to be sure you have good backups on either another storage card or your desktop by using the Newton Connection Kit early and often. ViewFrame is a powerful package.

What can you do with ViewFrame? The answer to that depends on how brave you are (or how recent your backups are).

ViewFrame's basic operation is object browsing. You can use it to examine any object in the Newton. ViewFrame also allows you to modify writable objects by changing the value of slots or array elements, and adding and deleting slots. Of course you can also browse soups (even create sophisticated queries) and edit their contents as well.

In addition to the expected object browser and editors ViewFrame can decompile any CodeBlock into NewtonScript. There are a few limitations (this is version 1.0 after all). For example, conditional expressions aren't fully rebuilt into valid NewtonScript but rather left in a form that reflects the underlying bytecode language. Also nested function definitions aren't fully listed. In general the limitations are not a problem since the only legitimate purpose of decompiling a copyrighted program (such as the Newton OS) is education. You should also note that anything you learn (that's not officially documented) using ViewFrame on a specific Newton platform is most likely only relevant to that specific machine. It is also usually illegal to decompile a program for the purpose of "reverse engineering".

When you get tired of browsing through the MessagePad's ROM you can write some NewtonScript programs using the ViewFrame Editor and Programmer's Keyboard. While you can't write complete applications, you can learn quite a bit about NewtonScript. (For the many individuals who want a taste of NewtonScript before investing in the NTK, ViewFrame is the only game in town.)

But wait, there's more! You can use ViewFrame to display objects on the MessagePad without recourse to the Inspector. You can even have ViewFrame call your application when ViewFrame is being closed so your debugging code can shut itself off. Yep, you can finally debug communications programs!

Finally, like good ol' ResEdit, ViewFrame is extensible. You can add your own Programmer's Keyboard or ViewFrame Editor if those supplied are not to your liking. You can also write new object viewing and editing functions. This process is somewhat more complex, but if you are in a position where you need this functionality you also know enough to create it fairly easily.

By now you should be getting the idea. ViewFrame is a nearly indispensable development tool, tremendously educational, and a whole lot of fun. The manual is well written and includes a lot of information that is not presently available anywhere else.

The only thing that ViewFrame doesn't have is a NewtonScript tutorial, (nor does it contain any of the basic Newton documentation delivered with the Newton Tool Kit). Complete NewtonScript novices still need to gain access to at least the NTK documentation. (One obvious add-on product is a NewtonScript tutorial with exercises tailored to ViewFrame.)

I highly recommend ViewFrame to every Newton programmer. Get it, Use it, Whoop! 

ALL ABOUT THE ARM RISC CHIP

by Howard Oakley
EHN & DIJ Oakley
Howard@quercus.demon.co.uk

The ARM RISC Chip, a Programmer's Guide, by Alex van Someren & Carol Attack, 346 pp., Addison-Wesley, ISBN 0-201-62410-9, price \$34.95.

The CPU in the Apple Newton MessagePad is the ARM610, a novel RISC processor developed by ARM Ltd in the UK. This is the first and only book to describe the processor in detail, and its intended market is obvious by the cover flash proclaiming "As used in the Apple® Newton™".

After a foreword by Larry Tesler (Apple's Chief Scientist and an influential member of the board of ARM Ltd) written just a few days after the Newton launch, van Someren and Attack give

a fascinating historical introduction to explain how an otherwise obscure British firm came to design the processor used in the Newton. It may come as some surprise to those not intimately knowledgeable of the British market, but ARM's progenitor, Acorn, has for some years been selling personal computers which use the ARM610's ancestor RISC processors, primarily into the UK home and educational markets.

Much of the book is devoted to C and assembly language programming of the ARM6 series, using the ARM Software Development Toolkit. Chapters cover the architecture of members of the family, integer instruction set, aborts, exceptions and interrupts, architectural extensions such as the memory management unit, interfacing, and the floating point instruction set. There are several examples of ARM assembler routines, and a Mandelbrot computation shown in its original C and the assembler produced from compilation. Appendixes summarize instruction set mnemonics and assembler directives, and there is a short bibliography.

If you have tried to understand the ARM610 data sheet, then you will find this book both more detailed and comprehensible. Although it stops short of a complete explanation as to how the Apple-specified MMU is so useful in the implementation of object-oriented systems, the 15 pages covering it are copiously illustrated with diagrams and elucidate much of its function. There are also valuable insights into the modular design of ARM processors.

However, this book is likely to be of limited value to most Newton developers. Many pages are devoted to descriptions of units, such as the FPU, which are not included in the ARM610, and there is no mention of any of the ROM-based features of the Newton, such as NewtonScript. Indeed, the impression given of the ease of development using the ARM SDT may leave you

yearning for it rather than the NTK. These desires need to be tempered by the reality of having to reinvent the wheel every time you start coding, of course, and the loss of the productivity and compatibility benefits of NewtonScript.

Rumors also abound about the future of ARM processors in Newton devices to be launched this year and later. The imminent arrival of the PowerPC 603 processor, the first low-power member of that family, may lure Apple away from the ARM series. Hints from Apple about not being too processor-dependent suggest that at least one Newton device will use the 603. ARM's answer, the 7 series, is already available, so there may well be a mixture of different CPUs in future Newton systems.

Although by no means an essential for those writing Newton applications, van Someren and Attack have produced a clear, understandable and useful text which you will find excellent background reading. For those who need to go below NewtonScript to write card drivers or whatever, it is likely to be indispensable. 

For a more detailed look at the ARM 610's innards you can get a copy of the 110+ page ARM610 Microprocessor data sheet from GEC Plessey Semiconductor. They have customer service offices in Swindon, Scotts Valley, Les Ulis Cedex, Munich, Milan, Tokyo, Singapore, and Stockholm.

Howard Oakley is a full-time Mac and Newton developer, and a part-time medical researcher. His development company has a range of specialist CAD/CAM applications in use primarily in the sailmaking industry. He wrote one of the first Newton freeware utilities, and has several commercial applications in early development.

Tap into Bear River for Newton™ Development and Training

Bear River is pleased to announce the expansion of its services to include . . .

- Newton Programmer Training
- Newton Software Development
- Systems Integration

We also offer development for Macintosh™ and Windows®, consulting, and programmer training for Macintosh.

Call Us Now!



Bear River Associates, Inc.
PO Box 1900, Berkeley, California 94701-1900
Telephone: 510-644-9400
Fax: 510-644-9778
AppleLink: BEARRIVER
Internet: marketing@bearriver.com

**COURSES
AVAILABLE
NOW**

GETTING STARTED

AN INTRODUCTION TO NEWTONSCRIPT MEMORY MANAGEMENT

Paul Potts
Pharos Technologies
potts@pharos-tech.com

This article discusses the rudiments of Newton's dynamic memory system. If you are accustomed to working in another memory environment, such as the C language's `malloc()`, or the Macintosh ToolBox's pointers and handles, you may find that getting used to the Newton chiefly requires unlearning habits and concerns picked up in these other environments. I show what habits are no longer necessary in the Newton environment, and what new habits help you handle Newton memory successfully.

To use the examples in the article you need to use the Newton Toolkit's Inspector window and a tethered Newton. Before we get started, I'd like to thank Mike Engber of PIE DTS for his feedback on drafts of this article. As always, any errors that have crept into this article should be considered the fault of the author and not the articles' reviewers or editor.

MEMORY BASICS

To start, let's find out how much free memory is available in the NewtonScript runtime environment. Type the following command into the Inspector window (press the <enter> key to execute the line; to execute multi-line entries, select all the lines and press the <enter> key):

```
Stats()
```

The response you get back depends largely on what you have in your Newton and what it is currently doing in addition to supporting the Inspector connection. On mine I get a response like the following:

```
Free: 34472, Largest: 34360  
#21AA0 34472
```

The `Stats()` function prints the amount of free space (in bytes), the size of the largest contiguous piece of memory available (in bytes), and then returns the total free space. (The second line is the function's return value. All NewtonScript functions have a return value. If you don't explicitly return a value with a `return` statement, the return value is the value of the last statement in the function).

Note that this free space concept doesn't mean all that much. The Newton's memory is broken into many parts:

- There are *stores*, where soup entries and packages are kept (the free space in the internal store is displayed when you look in Preferences under Memory). Putting a memory card in your Newton gives you another store (you can see the available space on this store by using the Card application in the Extras drawer).
- Next, there is the *NewtonScript heap*, where your application's runtime frames, and any frames and arrays allocated by executing NewtonScript code, live. This article concerns itself chiefly with the NewtonScript heap, which is set in the current MessagePad to 90K.

- There are operating system objects, written in C and C++, that are kept in their own *system heap*. This is the memory that has run out or become fragmented when handwriting recognition fails under early Newton system revisions. This world is pretty much off-limits to Newton developers.

THAT RUN-DOWN FEELING

Memory in the NewtonScript heap is precious and should be rationed carefully. The following function allocates arrays, using the `Array()` function, and sets each element to the value 9. It allocates an array of one thousand entries, ten times. Note that I don't bind the arrays to a name. Their return value, a pointer to the array, which would normally be stored in a slot and used to reference the array, is simply discarded. To put it in NewtonScript terms, no *references* remain to the arrays. Keep this in mind – it's an important distinction which affects garbage collection.

```
runDownMemory := func()  
begin  
  for z:=1 to 10 do  
    begin  
      Array(1000, 9);  
      Stats()  
    end;  
  return TRUE // I like explicit  
  // return statements for functions.  
end
```

Type in the lines of the `runDownMemory` function, then select them all and press Enter. You should get a response like the one below:

```
#44125F1 <CodeBlock, 0 args #44125F1>
```

This means that your code has compiled successfully into NewtonScript bytecode, and your function is stored in a newly-created slot called `runDownMemory`. You can execute this function with the following Inspector entry:

```
call runDownMemory with ()
```

An alternate method is to create the function as a global. If you define the function like this:

```
func runDownMemory ()  
begin  
  <body of function>  
end
```

you create a global function, which goes in the global functions frame. You can call this function from anywhere in the system with the syntax

```
runDownMemory()
```

Normally, creating a global function is not recommended, because it pollutes the global name space. However, global functions can be convenient while hacking around in the Inspector.

If you create a global, you should kill it after your are done by removing the slot it's kept in:

```
RemoveSlot(functions, 'runDownMemory')
```

Rebooting the Newton also removes the slot and the function stored in it.

Here are the results of my call to `runDownMemory` (with a few lines left out to save space):

```
Free: 28560, Largest: 28352
Free: 24544, Largest: 24336
<...>
Free: 4464, Largest: 4256
Free: 448, Largest: 240
Free: 32288, Largest: 32288
Free: 28272, Largest: 28272
#1A      TRUE
```

Observe the way that the amount of free memory changes. It is quickly run down, gets down to a low of 448 bytes, and then another call to `Array()` is done. Suddenly, it shoots back up to 32288.

The secret to this behavior is garbage collection. On the Newton, *garbage* is data that is no longer referenced by anything. When the Newton tries to allocate memory for data, if there is not sufficient space to do so, the global function `gc()` (garbage collect) is called by the system. If `gc()` frees up enough space for the requested memory, things continue as normal. If `gc()` can't free up enough space, the system generates an *exception*.

THE GARBAGE COLLECTOR GOES ON STRIKE

Let's make a small change to `runDownMemory`. We can simply replace `runDownMemory` with a new function. The reference to the old version is superseded and (you guessed it) the old function is eventually collected as garbage.

```
runDownMemory := func()
begin
    local a := Array(10, NIL);
    for z := 0 to 9 do
    begin
        a[z] := Array(1000, 9);
        Stats()
    end;
    return TRUE
end;
```

On my MessagePad, executing this method yields the following:

```
Free: 12040, Largest: 10536
Free: 8024, Largest: 6520
Free: 4008, Largest: 2504
Free: 19896, Largest: 19896
Free: 15880, Largest: 15880
Free: 11864, Largest: 11864
Free: 7848, Largest: 7848
Free: 3832, Largest: 3832
Exception |evt.ex.outofmem|: (-48216)
Run out of Frames Heap memory
```

A quick postmortem analysis shows what happened. The first three allocations proceed correctly. After the third array is allocated, the fourth allocation attempt probably fails and forces a `gc()`, which successfully frees enough memory to allow a successful allocation to occur and the function to continue. However, available memory eventually runs down, until a `gc()` fails, causing an exception.

Note that doing a `gc()` before each allocation call doesn't help – the modified function still causes an exception if frame heap memory is exhausted. So even if our loop looks like this:

```
for z := 0 to 9 do
begin
    gc();
    a[z] := Array(1000, 9);
    Stats()
end;
```

our allocation still fails as it did above because we are now keeping a reference to each array we allocate, stored in the array `a[]`. While that reference exists (which, in this case, is until `runDownMemory` finishes executing and local variable `a` is automatically set to `NIL`), the garbage collector cannot reclaim the space that the reference refers to.

You can use this knowledge to help the system do its job. Setting a reference to an object to `NIL` effectively severs the link between the reference and the object. `gc()` can then free the object's space (assuming there is no other live reference to the object). Add the following line to the `runDownMemory` function just after the call to allocate `a[z]`:

```
a[z] := NIL;
```

If there is enough memory available for one of the allocations to succeed, the function does not fail, because after each allocation we destroy the reference to the created array. This has the same effect on memory as our first version of `runDownMemory`, where we do not save a reference to the returned array.

COPING WITH FAILURE

Given the above, our situation as programmers might look rather hopeless. Since the system is so dynamic, it doesn't look like we can easily predict ahead of time if a call to allocate memory is going to succeed. We don't want our software to fail, however. How can we allocate memory when we don't know how much memory we have available to allocate?

Actually, the Newton has a powerful mechanism built in to help us cope with situations like this. We can go ahead and allocate all the memory we want to, as long as we consider in advance the consequences of failure. On the Newton, a failed allocation causes the system to raise an exception. The `try/oneexception` mechanism lets us plan for that type of exception and decide in advance what to do with it.

Consider this example:

```
SafeRunDownMemory := func()
begin
    try
        local a := Array(100, NIL)
        oneexception |evt.ex.outofmem| do
            begin
                print ("Error! Ran out of critical memory.");
                Rethrow() // We want our application to fail here
            end;
            local z := 0;
            try
                foreach item in a do
                    begin
                        a[z] := Array(1000, 9);
                        Stats();
                        z := z + 1
                    end
                oneexception |evt.ex.outofmem| do
                    print ("Error: ran out of memory allocating arrays.");
                return z
            end
        end
```


In this example we use two different `try/onexception` sets for different parts of our code. Suppose we consider the first allocation (setting aside 100 slots for the array `a` too critical to fail – if this allocation fails, our entire program should fail. If the `|evt.ex.outofmem|` exception occurs in this line, it is handled immediately below. Since we don't want this handler to "swallow" this exception, however, we immediately do a `Rethrow()` in the hope that someone else catches the exception. If no one else does, the user sees a familiar error dialog.

The second set of allocations, we know, is likely to fail. We may wish that we have enough memory to allocate a hundred arrays of a thousand objects, but we are realistic enough to know that we might not get our wish. In this case the `onexception` is likely to serve as the terminating condition of our `foreach()` loop. The counter `z`, which we return from the function, indicates the number of successful allocations. (After acquiring `z`, we should probably shrink our array of arrays to accurately reflect its true size, and to make certain that we don't attempt to access an element which was not successfully allocated).

Note that there are many other types of NewtonScript exceptions. In this example, I show some rudimentary methods for catching just memory allocation problems. A robust application design tests for and properly manages all raised exceptions, and a good programmer should include ways to recover, if possible, from most exceptional conditions.

SUMMARY

All this leads up to a few simple axioms:

You don't need to take out the trash yourself. As we have seen, NewtonScript collects garbage for you in order to allocate memory. Calling `gc()` yourself only slows down your code – it's a relatively expensive call. Allocations that would have succeeded after calling `gc()` always succeed anyway if you don't call `gc()`.

Local slots inside functions are set to NIL automatically when the function exits. You generally don't need to pay special attention to object references you place in these slots. The references are cut loose and the objects automatically garbage-collected when the system needs to reclaim space.

You need to set references to other objects to NIL in order for the garbage collector to reclaim their space. If you put references to objects in other frames, such as your application's base view frame, or the view system's root view, make sure you set to NIL any references that you don't need to keep around when your application closes. This allows the system to reclaim as much space as possible. Setting soup cursors to NIL is particularly important – this helps avoid the dreaded "Grip of Death," where the Newton doesn't allow you to eject a memory card.

The way to handle memory errors is to handle memory exceptions. Exceptions are the system's way of handling errors. Use them. It might be possible for you to do a `gc()` and then check for available space (using the return value of `Stats()`) before allocating some object, but this is not a reliable method – the Newton memory environment is very dynamic, and there are many things going on behind your back that can change available memory. In addition, you can't necessarily predict an object's size before allocation. Some objects, such as soup entries, are automatically compressed by the system. Even objects such as arrays may vary in size from system to system, particularly as Newton technology migrates to different hardware. Let the system tell you when an allocation fails – that's one of the reasons it's there.

Distinguish between recoverable and unrecoverable memory failures. In some cases, when an allocation fails, you might want to simply tell the user that they won't be able to do what they want, and ask them to try to do something a little less ambitious. In other cases, you won't be able to get the memory you need to do much of anything – your only alternative is to inform the user of that and gracefully quit. Let the user run the program and tell you what they want to do, then tell them when you can't do it. Don't make the user feel helpless by simply failing with no explanation or options, if you can help it.

Practice, practice, practice. With Newton memory management, Apple has eliminated most of the grunt work and potential for errors. I hope this article helps clear up a few of the misconceptions surrounding Newton memory management, and provides a little useful information about why the system behaves as it does. There is no substitute for practice, though. Start coding! Try out your ideas, and see for yourself how best to minimize your application's memory footprint and make efficient use of system resources. Good luck, and drop me a line at potts@pharos-tech.com if you come up with any good ideas that you would like to share. 🐸

Paul R. Potts is a software engineer who just left the University of Michigan at Ann Arbor to take a job with Pharos Technologies in Cincinnati, Ohio. When not hacking on the Newton he enjoys Tuvan-style overtone singing, running, and staring at the walls of his new, pretty-darned-empty apartment. Email him furniture, small appliances, or food at potts@pharos-tech.com.

The first postmodern case.

- Padded Cordura
- Screen protection
- Compact yet spacious
- Handle & shoulder strap



\$49.00 plus \$4.50 s/h

Fax } for { fax.
Call } info { mail.
Email } via { email.
Write } verbal pitch.

Newt Boot

Box 1613
Jackson WY 83001
307 733-0906
fax 739-1716
cis 72377,2740

AN INTRODUCTION TO VIEWFRAME

Jason Harper

76703.4222@compuserve.com

The debugging tools available for the Newton platform have been rather limited to date. Here are some of my experiences with the available tools, along with a description of my efforts to improve the situation.

THE INSPECTOR

The NTK (Newton Toolkit) has a built-in NewtonScript object examination tool called the Inspector. It runs on a host computer (currently, only a Macintosh) tethered to the Newton device being examined. This gives it the distinct advantage over any Newton-based tool of not being limited in its displays to the rather small screen size of current Newton devices.

However, it has some major limitations in the information it shows about the objects it displays, and in the options available for further displaying and modifying these objects. As an example, let's look at the output the Inspector might produce when we examine the `_proto` slot of an unidentified view:

```
#28F {_proto: {#31B},
      viewBounds: {#2963E5},
      viewJustify: 166,
      icon: {#4DF},
      buttonClickScript: <CodeBlock, 0 args #296405>}
```

What is this? From the fact that the frame has both an icon and a `buttonClickScript` slot, you might deduce that this is a `protoPictureButton`, or some variation thereof. Looking at the object's prototype might confirm or deny this. To do so, you have to retype the reference number of the `_proto` slot, in the form `ref(0x31b)`. [According to DTS, *Ref* is an unreliable function that will be "going away soon." They advise against its use. Ed]. In this case, the prototype contains a `debug` slot with the value "protoPictureButton" (many of the system proto templates have a `debug` slot, so you can find out the basic type of many views if you follow their `_proto` chain far enough).

However, it is not always possible to examine frame slots in the Inspector. There are two reasons why your peek might fail:

- The reference may no longer be valid. The NewtonScript garbage collector may have reclaimed or relocated the object. The garbage collector can hardly be expected to notice that there is still a reference to the object, on the screen of another computer.
- It may be impossible to reenter the reference value. NewtonScript integers are limited to 30 bits, but references are full 32-bit values. If either of the highest two bits of a reference are set, there is no way to directly enter it. You have to use a workaround such as specifying a different path to the desired slot, such as `ref(0x28f)._proto` in the current example.

We've identified the object as a picture button, but what button is it? Looking at its icon might tell, so let's do so:

```
#4DF {mask: <mask, length 52>,
      bits: <bits, length 52>,
      bounds: {#235F55}}
```

This is, unfortunately, about as far as the Inspector can go. You can look at the `bounds` slot to find that the icon is 9 by 9 pixels, but the Inspector has no capability for showing any meaningful representation of the `mask` and `bits` slots, or any other binary objects that don't have a defined textual version. You can get a little more information by going back to the object's

`buttonClickScript` – typing in `ref(0x296405).literals` reveals that the script uses the symbols `base` and `Close`. Perhaps the script is `base:Close()`? There's no way to tell for sure.

OBJVIEWER

It wasn't very long after the Newton introduction that a stand-alone NewtonScript object examining tool appeared – `ObjViewer`, a freeware program by Robert P. Munafo which was discussed in the last issue of *PIE Developers*. It has a movable window, half the height of current Newton screens, so you can always see what else is going on. It displays ten lines at a time of frames, arrays, binary objects (shown in hexadecimal) and other data types. Viewing an object referenced by a frame or array slot is as simple as tapping on the line describing it – there is no chance of failure as with the Inspector.

For comparison, here is our example frame as displayed by `ObjViewer`:

```
Frame!0000028F:0/5 slots
_proto: {..}
viewBounds: {..}
viewJustify: 166
icon: {..}
buttonClickScript: {..}CodeBlock
```

Note that frames are displayed with the number of slots they contain, generally a far more useful piece of data than the reference number shown in the Inspector. The `icon` object can be viewed more easily than before, and we can even look at its data bits:

```
bits!00235FB6:52 bytes
0: 000000000000401C5I
8: 01C5025E01CE0267I ^ g
16: 1C800000E3800000I
24: 770000003E000000w >
32: 1C0000003E000000I >
40: 77000000E3800000w
48: C1800000I
```

Unfortunately, this still doesn't tell us what the view actually is, unless you're a lot better at visualizing binary data than I am.

A major deficiency of `ObjViewer` is that it can only examine objects reachable from the root view, as returned by the `GetRoot()` global function. This is the root of the view system only, not of the entire Newton object system. There are many items of interest that can't be reached from `GetRoot()`, or that don't exist as objects at all until a program requests them (such as soup entries). Also, `ObjViewer` is a purely read-only tool. Sometimes it is more convenient to test a small program change directly in the Newton memory rather than taking the time to recompile the whole program.

INSPECTOR GADGET

Another tool that developers may find useful, which may seem unrelated to object viewing at first glance, is a way to test fragments of NewtonScript code without the time and trouble required to set up a complete program in the NTK. The Inspector can be used for this purpose, although it can be somewhat tricky: some NewtonScript constructs don't work directly from the Inspector – they have to be enclosed in a function which is then called. Also, the Inspector isn't always available.

What if you are miles away from your Mac, and are suddenly just dying to know what happens if you remove a slot from a frame you're currently looping through with a `foreach` statement? Apple supplies a simple NewtonScript experimentation tool named `Inspector Gadget`. It's one of the sample programs included with the NTK. A variation on this program has

been distributed by Howard Oakley under the name NewtonScript Runner. While these tools can be useful (I did a lot of the early work that led to ViewFrame using Inspector Gadget, since I didn't have a Macintosh of my own at the time), they are rather clumsy to use, due to the difficulty of entering NewtonScript expressions directly on a Newton device.

Handwritten input is impractical – several important NewtonScript punctuation marks, such as curly and square brackets, are not recognized no matter how clearly you write them. The on-screen keyboard works, but it's constantly asking you if you want to add useless words to your personal dictionary, and it's rather slow to use. Tap in "viewSetupFormScript" a few times, and you know what I mean.

But perhaps the main problem with these utilities is their limitation in displaying the results from code you entered. Both just turn the result into a string with the `SPrintObject()` function, which returns an empty string for many objects. All frames and arrays, as well as some simpler objects such as `NIL` and `TRUE` simply come out blank. It's possible to enter a `foreach` loop that turns frames and arrays into meaningful strings which can be returned, but this routine takes up much of the input area provided by these tools, leaving little room for anything for the routine to operate on.

VIEWFRAME

Faced with a desire to learn more about how my Newton MessagePad works than was documented, and a lack of really good tools to do so, I decided to write my own object browsing utility. I wanted something that could meaningfully display any NewtonScript object, and had no limitations on what objects could be viewed. This program, which ended up being called ViewFrame, is obviously going to be a work-in-progress for as long as Apple continues to refine the Newton system. But even in its first version it is quite capable of showing you more of what's going on "under the hood" than any existing tool.

Figure 1 below shows how ViewFrame displays the unidentified frame of the earlier examples:

- Note that the debug slot's value ("protoPictureButton") from the object's prototype is shown in the listing for the `_proto` slot, saving you the effort of opening that slot to see what it is.
- Notice the `@nnn` notations that appear in several places. These are magic pointers, an important NewtonScript construct that is unfortunately discussed only briefly in current documentation. By looking them up in the NTK Definitions file you can tell exactly what the referenced object is. In this case the `_proto` is an object named `protoPictureButton` (already identified by the debug slot),

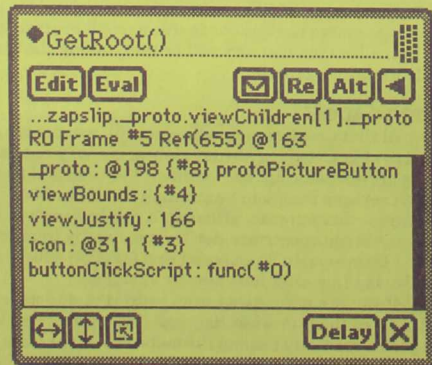


Figure 1 - ViewFrame's main window

and the icon is named `ROM_cancelbitmap`. The object itself is finally identified as the `protoCancelButton`.

Hint: you can identify magic pointers using other debugging tools, too. Subtract three from an object's reference value, then divide by four. If the result is an integer, that's the magic pointer number for the object. This is probably not guaranteed to remain true for future Newton devices, but it's a quite useful trick for now. *[There is also a function called `IsMagicPtr`. Ed].*

Figure 2 shows more examples of ViewFrame's object displays, from slots in the `protoCancelButton` frame. Icons and pictures are displayed in graphical form, and currently eight common types of flags fields can be displayed as symbolic constants. Even NewtonScript functions can be displayed as an approximation of the original source code, although the results aren't always as accurate as in the example (the current version of ViewFrame can't fully interpret looping and conditional constructs).

OTHER FEATURES

ViewFrame has quite a few other features:

- In addition to the object types shown in the examples, there are specific view formats for sounds (which are played), soup entries, strings and other types. Many object types have an alternate view format. For example, the icon above can be viewed as its component binary objects, if you really want to see that information.
- ViewFrame's window is fully resizeable, so you can have it take up the whole screen if you don't need to see anything else at the same time. If the full screen isn't enough area, you can dump the output to the Inspector if it's connected.
- The contents of Newton dictionaries (other than lexical dictionaries, such as the phone numbers recognizer) can be dumped to the Inspector.
- You can navigate the Newton object hierarchy by either tapping on slots or entering expressions at the top of the window. Tapping the diamond in the upper lefthand corner of the ViewFrame window brings up a list of expressions you can use, including common places to start browsing such as `GetRoot()` and `GetGlobals()`, and expressions relevant to the current object such as `:ChildViewFrames()`, when the object is an open view. Almost any NewtonScript operation can be executed using ViewFrame by entering the proper expressions. For example, by using the functions `GetStores()`, `:GetSoup()`, `Query()`, `:Entry()` and `:Next()`, all of which appear in the expression list at the appropriate time, you can examine the contents of any soup.
- Programs that you write can call ViewFrame directly to display an arbitrary object, using a statement such as:

```
GetRoot().[ViewFrame:JRH]:?NewValue(
    description, object);
```

It's hardly as convenient as using a simple `print()` statement to send output to the Inspector, but can be used at times when the Inspector is unusable (such as while debugging a program that uses serial I/O).

ACCESSORIES

In order to make it easy to experiment with NewtonScript code, I wrote a companion program called the ViewFrame Editor (see Figure 3). It looks and acts much like the Newton Notepad, except that drawn shapes are not allowed. The returned value from programs you enter can be displayed using ViewFrame, so you don't have to worry about formatting the return value in a human-readable form. The Editor supports scrolling, so program size is limited only by available memory.

In order to make NewtonScript entry practical in the ViewFrame Editor and the expression entry area of ViewFrame itself, I also created a Programmer's Keyboard (shown in Figure 3). It works the same as the standard on-screen keyboard (except that entry of words into your personal dictionary isn't supported), but also has several common NewtonScript phrases and punctuation marks available for entry with a single tap. For example, "viewSetupFormScript" can be typed in with only seven taps: "view", "Setup" and "Script" require only one tap each. As an added bonus, the Keyboard can be switched to a Dvorak layout for those of you who are more comfortable with that.

AVAILABILITY

As I mentioned, ViewFrame is a work in progress. By the time you read this, current owners should have already received a free editor upgrade. In addition, there are a variety of new features that I'm working on. Judging by the response from the Newton developer community, it's a very useful utility. ☺

Jason Harper is a freelance Apple II programmer who still, somehow, has enough faith in Apple to give Newton development a try. His other Newton programs include ExtraExtras, an Extras Drawer extender, and BeamToSelf, for testing beaming with only a single Newton device.

ViewFrame is available exclusively from Creative Digital Systems. See the inside back cover of this issue for details. A demo version of ViewFrame is available on most major bulletin board systems and on the PIE Developers 2.2 source code disk.

Size: 9 x 9

X	viewJustify slot	
Mask:	◆ View as: viewJustify	
::	166	func() begin
Hilited:	0xA6	base:Close()
*	= vjCenterH	end
Icon slot	+ vjCenterV	buttonClickScript slot
	+ vjParentRightH	
	+ vjParentBottomV	

Figure 2 - Other ViewFrame object displays

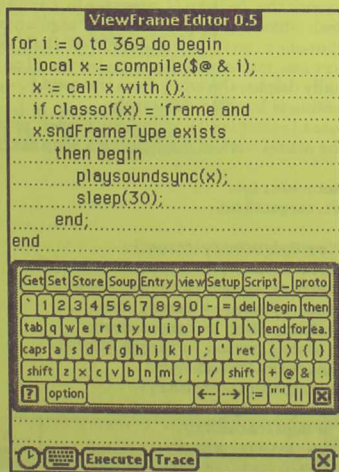


Figure 3 - the ViewFrame Editor and Programmer's Keyboard

BUILDING THE PERFECT BEAST II

Gregory N. Christie
theObjectPartnership
G.CHIRSTIE@applelink.apple.com

Well, it's been a long two months since the last issue of *PIE Developers*, and a lot has happened to the ExpenseMan application. This month I add a soup and handle some basic routing. The soup holds the user's expenses and the routing manages basic soup manipulations like deleting, duplicating and moving entries to and from cards.

Before I begin, let me recommend that you reread (or read!) three articles from previous issues of *PIE Developers*. First, you should go through the first article of this series, in issue 2.1, to refresh your memory on the ExpenseMan project. Second, Mike Engber's article "Soup's On" in issue 1.1 provides the necessary background information for using soups, cursors and entries. Finally, I rely heavily on the techniques Mike Engber describes in "Tales From the View System" also in issue 1.1. In "Building the Perfect Beast II" I assume that you have read these materials, and that you have a basic knowledge of soups, cursors, entries and views.

DESIGN

Most programming environments rigorously enforce the separation of a program's views and its data. Newton, being amazingly flexible, does not. The views you use in an application can actually be soup entries. This is quite different from the conventional model where you have a fixed view and shuttle data in and out of that view. This poses a interesting question for the developer, one that has to be answered for each Newton project: what do you keep in a soup - views, data or both?

As with all things Newt, the answer depends. If you use the soup's entries to hold an item's viewBounds or proto, in addition to the item itself, then those slots are applicable only to a single view of the data. If you only have a single view of the data, or have implemented an application as a Roll Browser, then go ahead and stuff those views into the soup. If, as in ExpenseMan, you have more than one view of the data, then it's probably better to separate the views from the data they show.

There is another soup-related design question that should be answered before a soup is added to the application: how should a blank soup be displayed? There are two basic approaches you can take. You can design an application to behave like the NotePad, where there is always at least one entry in the soup, or like the Names application, where the message, "There are no Names..." appears when a soup is empty. I use the NotePad example, creating a new entry whenever the last one is deleted.

As I explain the mechanics of adding soup support to ExpenseMan, notice that I register and unregister with system services when ExpenseMan opens and closes, rather than in the project's InstallScript() and RemoveScript(). There are three reasons for this. First, since ExpenseMan is under construction, I do not want it responding to system messages while closed. Secondly, I am a maniac about conserving heap space. Until ExpenseMan is complete, I will not put it completely "on-line." Finally, if you have bugs in your InstallScript() or RemoveScript() it can be extremely difficult to remove a misbehaving application from the Newton without giving it (or the card) a factory reset. I generally prefer to leave the system alone and test the Install and Remove functions from within an application until I am finished with a project and certain of its behavior. Throughout this article I indicate where my NewtonScript code would normally be placed in the project's InstallScript() or RemoveScript().

STIRRING THE SOUP

Following PIE DTS's signature guidelines, the soup for ExpenseMan is named after ExpenseMan's appSymbol slot, ExpenseMan:SIG. Remember that to use ":" in a symbol name in NewtonScript, the symbol must be enclosed by vertical bars ("|"). The soup has five slots—amount, when, desc, person and labels. The labels slot is reserved for when I add folders support to ExpenseMan. Keep in mind that a soup is not a hard-formatted database where you have to define all the fields before using it. You can add slots at will, whenever you or your application needs them. Pretty much the only restriction on entry slots is that an indexed slot should always contain the same type of data. My soup is indexed on the when slot, therefore I want to be sure that it always holds a time value, an integer representing the number of minutes since midnight, January 1, 1904.

ExpenseMan has four basic views:

- expenseMan, the base view;
- emEntryview, a single expense view;
- emOverview, the overview; and
- emExpenseLine, a single expense user proto used in emOverview.

Each of these views interact with the soup. The basic support is handled in the base view, expenseMan

EXPENSEMAN - THE BASE VIEW

When the base view is opened, the soup is set up and initialized. A few slots need to be created immediately, before the scripts which reference them are created. The slot emSoupName holds a string with the name of the soup. emSoup, emCursor and currentEntry are initially set to nil, but when ExpenseMan is running, they hold the current soup, cursor and entry. Several scripts handle the setup and initialization of the soup, including the ViewSetupFormScript():

```
ViewSetupFormScript:
func()
begin
  GetGlobals().routing.(appSymbol) :=
    EnsureInternal(routingFrame);
  emSoup := :RegisterCardSoup(emSoupName,
    emSoupIndices, appSymbol, appObject);
  emCursor := :MakeCursor();
  if NOT emCursor.current then
    begin
      local newEntry := :CreateNewEntry();
      emCursor.Goto(newEntry);
    end;
  currentEntry := emCursor.current;
  target := emCursor.current;
  :SizeAppToScreen();
end
```

This script does quite a lot of work. First it registers with two system services, routing and CardSoups. Registering with routing, a straightforward process, adds support to ExpenseMan's action button. (In a finished application this routine would probably appear in the InstallScript() in the Project Data file.) For the routing, ExpenseMan needs to handle deleting, duplicating and card transfers. To support these three functions, I must add several slots to the expenseMan view—a routing frame and some action scripts.

The Routing Frame

First of all, I need to add a routing frame. ExpenseMan's looks like this:

```
routingFrame:
(
  delete:{title: 'Delete Expense',
    routeScript: 'DeleteActionScript'},
  duplicate:{title: 'Duplicate',
    routeScript: 'DuplicateActionScript'},
  card:{GetTitle: func(item)
  begin
    if item AND Length(GetStores())>1 then
      if EntryStore(item) = GetStores()[0] then
        "Move to card"
      else
        begin
          if EntryStore(item):IsReadOnly() then
            "Copy from card"
          else
            "Move from card";
        end
      end
    nil;
  end,
  routeScript: 'CardActionScript'
)
```

This frame is added to the routing frame in the system's globals frame. To use it correctly, expenseMan must have slots called target and targetView in it's base view. They hold the current soup entry and view, respectively.

I also have to write the scripts DeleteActionScript() and DuplicateActionScript() which are referenced in the routing frame. The Newton itself implements a CardActionScript() to handle transferring items to and from cards. If you implement the card slot in the routing frame as I have, your application can inherit this functionality also.

Action Scripts

The DeleteActionScript is:

```
deleteActionScript:
func(target, targetView)
begin
  targetView:Delete('doDeleteAction, [target, targetView]);
end
```

This script sends the Delete() message to targetView (the current view), which shows the "crumple" animation and also calls DoDeleteAction() in the targetView. Both emEntryview and emOverview implement the DoDeleteAction() script which actually deletes the entry from the soup.

Duplication is handled differently.

DuplicateActionScript() is implemented in expenseMan, and does the actual work of duplicating a soup entry:

```
duplicateActionScript:
func(target, targetView)
begin
  targetView:UpdateEntry(target);
  local duplicateEntry := :CreateNewEntry();
  duplicateEntry.amount := target.amount;
  duplicateEntry.when := target.when;
  duplicateEntry.desc := target.desc;
  duplicateEntry.person := target.person;
  EntryChange(duplicateEntry);
  emCursor.Goto(duplicateEntry);
  currentEntry := duplicateEntry;
  target := duplicateEntry;
  targetView:UpdateFromEntry();
end
```


This script creates a blank entry, sets the blank entry's slots to match those of the current entry, writes the changed entry to the soup and updates the current view. With the addition of the routing frame and its associated slots, ExpenseMan supports an action button, which is added to the status bar at the bottom of expenseMan (see Figure 1). Now that routing support is available, ExpenseMan can delete, duplicate and transfer entries.

Soup Registration

Looking back at `ViewSetupFormScript()`, the second line registers `ExpenseMan`'s soup using `RegisterCardSoup()`. (This is another script which can be placed in the `InstallScript` of an application's Project Data file.) `RegisterCardSoup()` is shown below:

```
RegisterCardSoup:
func(soupName, soupIndices, appSymbol, appObject)
begin
    if functions.RegisterCardSoup then
        return RegisterCardSoup(soupName,
            soupIndices, appSymbol, appObject);
    CreateAppSoup(soupName, soupIndices,
        EnsureInternal([appSymbol]), EnsureInternal(appObject));
    AddArraySlot(CardSoups, soupName);
    AddArraySlot(CardSoups, soupIndices);
    local store;
    foreach store in GetStores() do
        if NOT store.IsReadOnly() AND NOT
            store.HasSoup(soupName) then
            store.CreateSoup(soupName, soupIndices);
    GetUnionSoup(soupName);
end,
```

`RegisterCardSoup()` creates a soup called `ExpenseMan:SIG` with an index as defined in the slot `emSoupIndices`:

```
emSoupIndices: [{structure: 'slot',
    path: 'when, type: 'int'}]
```

Because `emSoupIndices` specifies type `'int'` for the when slot, you must be careful when creating entries to ensure that the when slot always contains an integer. `RegisterCardSoup()` also ensures that the soup is created on every available store, and that if a new card is inserted while `ExpenseMan` is open, a soup is created there as well. Finally, `RegisterCardSoup` returns a `UnionSoup` which is stored in the `emSoup` slot.

The Soup Cursor

After the soup is created, a cursor must be created to facilitate getting entries from the soup. `MakeCursor()` handles this:

```
MakeCursor:
func()
begin
    if NOT emSoup then
        emSoup := RegisterCardSoup(emSoupName, emSoupIndices,
            appSymbol, appObject);
        local queryType := {type: 'index, indexPath: 'when'};
        return Query(emSoup, queryType);
    end,
```

`MakeCursor()` first checks to be sure that the soup is set up. Then it returns a cursor of all entries indexed on the when slot. After the soup and cursor are initialized, `ViewSetupFormScript()` makes sure that the soup is not empty. If it is, a blank entry is created and added to the soup. The script `CreateNewEntry()` does this:

```
CreateNewEntry:
func()
begin
    local blankEntry := {
        amount: 0,
        when: Time(),
        desc: "",
        person: "",
        labels: nil};
    emSoup.AddToDefaultStore(blankEntry);
    return blankEntry;
end
```

After all this setup, the slots `currentEntry` and `target` are set and `ViewSetupFormScript()` exits.

The New Button

The base view `expenseMan` contains a status bar which has an action button (the routing slip) and a "New" button. No additional scripting is needed for the action button. The Newton builds its popup menu from the information in `ExpenseMan`'s routing frame. The following script is added to the "New" button:

```
buttonClickScript:
func()
begin
    if currentEntry then currentView.UpdateEntry();
    local theNewEntry := createNewEntry();
    emCursor.Goto(theNewEntry);
    currentEntry := theNewEntry;
    target := theNewEntry;
    currentView.UpdateFromEntry();
    :DisplayEntryview();
end
```

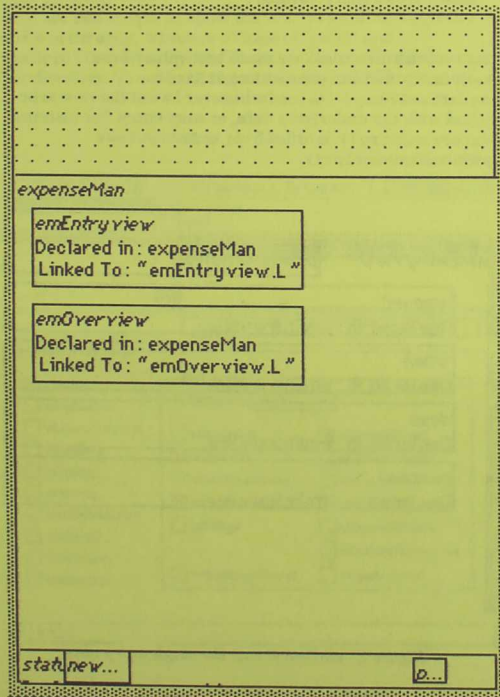


Figure 1 - expenseMan, the base view

The `ButtonClickScript()` updates the current entry, creates a blank one, and opens `emEntryview` so that the user can record the information for the new expense.

Finishing Up

Before describing `emEntryview` and `emOverview`, I want to show the base view's `viewQuitScript`:

```
func()
begin
  RemoveSlot(GetGlobals().routing, (appSymbol));
  :UnRegisterCardSoup(emSoupName);
  emSoup := nil;
  emCursor := nil;
  currentEntry := nil;
  expenseMan := nil;
  target := nil;
  targetView := nil;
  currentView := nil;
end
```

This script first removes `ExpenseMan`'s routing frame from the Newton's global routing frame then unregisters the soup with the system:

```
UnRegisterCardSoup:
func(soupName)
begin
  if functions.UnRegisterCardSoup then
    return UnRegisterCardSoup(soupName);
  local pos := ArrayPos(CardSoups, soupName, 0,
    func(x,y) ClassOf(y)='String AND StrEqual(x,y));
  if pos then ArrayRemoveCount(CardSoups, pos, 2);
end
```

(These two unregistrations would typically be done in the `RemoveScript()` in the application's Project Data file.) Once the soup is unregistered, I set to `nil` all slots which reference soups, cursors, entries and views. This is so the system can reclaim the considerable memory used by these types of references. When you venture into the wilderness, and the Newton is no exception, you should always pack out what you pack in.

For now, `ExpenseMan` always opens in the overview, and does not automatically redisplay the last entry you worked on. When `ExpenseMan` is finished, I will add a few lines of code to add an entry to the system soup to preserve this information. As I mentioned earlier, until I finish an application, I prefer to keep my hands off the system. When `ExpenseMan` is complete, I will also add application-wide support for Undo, another service I typically leave for last.

EMENTRYVIEW - THE SINGLE EXPENSE VIEW

`emEntryview` displays a single soup entry. It allows the user to make changes to the entry and includes several customized `protoLabelInputLines` (see Figure 2). There are two methods defined in `emEntryview` which are used constantly, `UpdateEntry()` and `UpdateFromEntry()`.

Update Methods

`UpdateEntry()` writes any changes the user makes back to the soup. It is called whenever there is a possibility that the view has changed:

```
UpdateEntry:
func()
begin
  base.currentEntry.amount :=
    StringToNumber(amount.entryLine.text);
  base.currentEntry.when :=
    StringToDate(when.entryLine.text);
  base.currentEntry.desc := desc.entryLine.text;
  base.currentEntry.person := person.entryLine.text;
  entryChange(base.currentEntry);
end
```

For each slot in the current soup entry, `UpdateEntry()` makes sure that the current value of the corresponding `protoLabelInputLine` is formatted correctly and placed in the entry. I use `StringToNumber()` and `StringToDate()` to convert text entries into numbers for the soup.

`UpdateFromEntry()` does the reverse. It takes the slots in the current entry and formats them correctly for the view's `protoLabelInputLines`:

```
UpdateFromEntry:
func()
begin
  SetValue(amount.entryLine, 'text',
    "$"&FormattedNumberStr(
      base.currentEntry.amount, "%.2f"));
  SetValue(when.entryLine, 'text', DateNTIME(
    base.currentEntry.when));
  SetValue(desc.entryLine, 'text',
    base.currentEntry.desc);
  SetValue(person.entryLine, 'text',
    base.currentEntry.person);
end
```

To convert the numerical slots into text values I use

`DateNTIME()` and `FormattedNumberStr()`.

`UpdateFromEntry()` is used whenever I want the view to be synched with the underlying data, or soup entry. For instance, `UpdateFromEntry()` is called from `emEntryview's` `ViewSetupDoneScript()`.

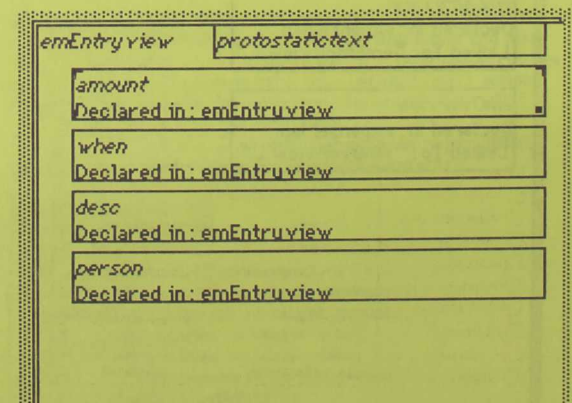


Figure 2 - `emEntryview`, the single entry view

Soup Entry Deletion

In addition to these two scripts, I implement the `DoDeleteAction()` which is called by the `DeleteActionScript()` in the base view. This is the script that actually does the work of deleting an entry from the soup:

```
dodeleteAction:
func(entry,entryView)
begin
    entryRemoveFromSoup(entry);
    emCursor:Next();
    if NOT emCursor.current then
        emCursor:Prev();
    if NOT emCursor.current then
        emCursor:Reset();
    if NOT emCursor.current then
        begin
            local theNewEntry := :createNewEntry();
            emCursor:Goto(theNewEntry);
        end;
    base.currentEntry := emCursor.current;
    base.target:= emCursor.Current;
    entryView:UpdateFromEntry();
end
```

`DoDeleteAction()` deletes an entry and advances the cursor to the next entry. If the next entry is `nil` (the end of the cursor), then the cursor is backed up to the last entry. If this entry is `nil`, I try to reset the cursor to the beginning. If that entry is `nil`, then the soup is empty, and I create a blank entry. The view is then updated from the entry.

Soup Scrolling

There are two scripts to allow the user to scroll through the entries in the soup, `ViewScrollUpScript()` and `ViewScrollDownScript()`. These scripts are identical except for their direction. `ViewScrollUpScript()` looks like this:

```
viewScrollUpScript:
func()
begin
    :UpdateEntry();
    playSound(ROM_flip);
    if not emCursor:prev() then
        begin
            emCursor:next();
            playSound(ROM_funBeep);
        end;
    base.currentEntry := emCursor.current;
    base.target:= emCursor.Current;
    :UpdateFromEntry();
end
```

First, any changes made to the entry are updated in the soup. Then the flip sound plays as the cursor moves to the previous entry. If the cursor is past the start of the soup, the FunBeep sound plays and the cursor moves to the first entry in the soup. The view is then updated from the current entry.

Data Entry

`UpdateEntry()`, `UpdateFromEntry()`, `DoDeleteAction()`, `ViewScrollUpScript()` and `ViewScrollDownScript()` manage the basics of handling single soup entries and their representations in a view. The individual `protoLabelInputLines` in `emEntryview` are customized to reflect the types of data in each slot.

Recognition in the amount, when, desc and person `protoLabelInputLines` is restricted by setting the `entryFlags` and `viewFlags` slots in the NTK. Be sure to set the class for these slots to match the data types in their corresponding soup slots. For instance, set the class in the `viewFlags` and `entryFlags` slots of the when `protoLabelInputLine` to "Date" (see Figure 3).

In addition to these slot settings, I create a "smart" `TextChanged()` script in the name `protoLabelInputLine`. This script is called whenever the user writes in the name `proto`. It sets up a popup menu that the user can pick from that contains a list of matching names from the Newton's Names soup:

```
textChanged:
func()
begin
    local theName := TrimString(entryline.text);
    local commandArray := [];
    if theName AND NOT StrEqual(theName, "") then
        begin
            local nameArray := SmartCFQuery(theName);
            If nameArray then
                foreach nameFrame in nameArray do
                    AddArraySlot(commandArray, nameFrame.name.first &&
                        nameFrame.name.Last);
                end;
            :setLabelCommands(commandArray);
            :UpdateEntry();
        end
    end
```

The script gets the text from the `name` entry line and removes any leading or trailing spaces using the `TrimString()` function. A blank command array for the popup menu is created. If the text from the `protoLabelInputLine` is not empty I call `SmartCFQuery()`. `SmartCFQuery()` which returns an array of frames containing the information from entries in the name soup which match the text in the `protoLabelInputLine`. I iterate over this returned array and build the command array for the popup menu. A slot in the command array contains the first and last names for a name returned from the `SmartCFQuery()`.

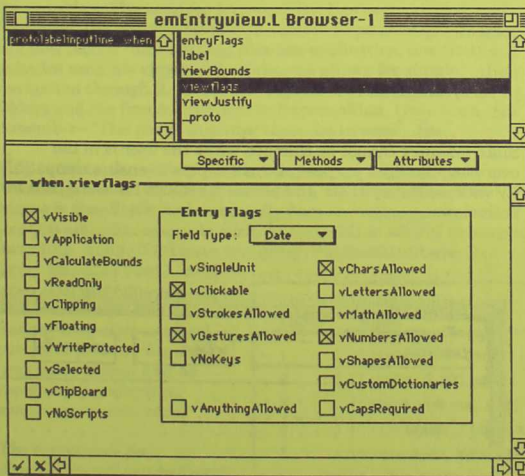


Figure 3 - the Browser window

Once the command array is built, I call the `SetLabelCommands()` method. The Newton handles the details of displaying the popup menu when the user taps on the diamond, and updating the `protoLabelInputLine`'s text when an item is chosen. By including this feature in the name `proto`, `ExpenseMan` becomes much more of a Newton-like application and it behaves as the user would expect. You can add a similar "smart" `TextChanged()` script to the `when` `protoLabelInputLine` to handle dates specified as "Today", "Tomorrow" or "Friday."

EMOVERVIEW

The overview for `ExpenseMan`, `emOverview`, is a simple list of all entries in the soup. Each expense is shown on a single line by a view defined by the user `proto emExpenseLine`. Basically, the overview has to dynamically create and destroy these views as the soup changes. This is accomplished by changing `emOverview`'s `stepChildren` slot using several interrelated methods: `ViewSetupDoneScript()`, `UpdateFromEntry()`, `CreateViews()` and `CreateAView()`.

View Management

First, here's the `ViewSetupDoneScript()`:

```
viewSetupDoneScript:
func()
begin
    if NOT HasSlot(self, 'stepChildren') then
        self.stepChildren := Clone(self.stepChildren);
        :UpdateFromEntry();
    end,
```

Since I dynamically create and destroy views by manipulating `stepChildren`, I need to be sure that `emOverview`'s `stepChildren` array is located in RAM. See Mike Engber's "Tales From the View System" in *PIE Developers* 1.1 for a thorough explanation of this technique.

Once the `stepChildren` array is created, I call:

```
UpdateFromEntry:
func()
begin
    RemoveSlot(self, 'stepChildren');
    self.stepChildren := Clone(self.stepChildren);
    :CreateViews();
    if length(self.stepChildren) > 7 then
        ArrayRemoveCount(self.stepChildren, 7,
            (length(self.stepChildren)-7));
        :RedoChildren();
    end
```

`UpdateFromEntry()` destroys the `stepChildren` slot in `emOverview` and constructs a new one. If `UpdateFromEntry()` is only called from the `ViewSetupDoneScript()`, this would be unnecessary. However, it is called from several other scripts, so it must be as general as possible. `CreateViews()` is called to create the actual `stepChildren`. Finally the length of `emOverview`'s `stepChildren` array is checked. (Since only six `emExpenseLines` fit in the overview at one time, only that many are kept in the `stepChildren` array.) Finally, `RedoChildren()` is called to draw the views within `emOverview`.

`CreateViews()` iterates over the entries in the soup and creates a view for each one:

```
CreateViews:
func()
begin
    MapCursor(emCursor, func(entry) begin
        :CreateAView(entry); nil; end);
    end
```

`MapCursor()` applies a function to every entry in a soup's cursor. Here I am not concerned with the returned value of the `MapCursor()` call – I use it strictly to operate on every entry in the cursor. The function I apply is `CreateAView()`:

```
CreateAView:
func(theEntry)
begin
    local template :=
        (entry:theEntry, _proto:PT_emExpenseLine);
    AddArraySlot(self.stepChildren, template);
    end
```

`CreateAView()` should perhaps be called `CreateATemplate()`, since it actually creates templates and not views from my user `proto emExpenseLine`. As each template is created, it is stuffed into `emOverview`'s `stepChildren` array. When `RedoChildren()` is called in the `ViewSetupDoneScript()` the child views are actually created.

Scrolling, Deleting, and Updating

After the overview is created, a few scripts have to be created to handle scrolling and deleting: `DoDeleteAction()`, `ViewScrollUpScript()` and `ViewScrollDownScript()`. Believe it or not, these scripts are lifted entirely from `emEntryview`. This works because the "update view" script is called `UpdateFromEntry()`. In fact, in a finished application, I would move these scripts to the base view. For now, and for testing purposes, I prefer to have these scripts locally in `emOverview`.

One more script must be added to `emOverview` so that its interface matches `emEntryview`. That script is `UpdateEntry()`:

```
UpdateEntry:
func()
begin
    EntryChange(currentEntry);
    end
```

Since an entry cannot be changed while the user is in the overview, the chief purpose of this script is to ensure compatibility with `emEntryview` and allows me to use the same code for both views.

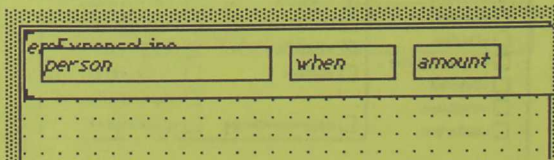


Figure 4 - the user `proto emExpenseLine`

EMEXPENSELINE

The user proto `emExpenseLine` is very simple. It contains three `protoStaticTexts` for the person, when and amount slots in the soup (see Figure 4). The proto also contains a slot called `entry` which is initially set to `nil`. This slot points to the view's corresponding entry in the soup, and is set in `emOverview's CreateAView()` method. The text slots in the `protoStaticTexts` are set in the `ViewSetupDoneScript()`:

```
ViewSetupDoneScript:
func()
begin
    SetValue(person, 'text', entry.person);
    SetValue(when, 'text', ShortDate(entry.when));
    SetValue(amount, 'text', "$"&FormattedNumberStr(entry.amount, "%.2f"));
end
```

The proto's `entry` slot is used here to set the text slots. `ShortDate()` and `FormattedNumberStr()` are used to format the text values for the proto from the numerical values in the soup entry.

To enable the user to switch to the single-entry view when a particular line is tapped in the overview, I use a `ViewClickScript()`:

```
ViewClickScript:
func(unit)
begin
    :TrackHilite(unit);
    base.emCursor:Goto(entry);
    base.CurrentEntry := emCursor.current;
    base.target := emCursor.current;
    :DisplayEntryView();
end
```

First, the tapped item is highlighted. When the user lifts the stylus, the `ViewClickScript()` sets that entry to the current entry and opens the entry view `emEntryView`.

PAST, PRESENT, FUTURE

Now I've added support for a soup and soup-related routing to `ExpenseMan`. The soup for `ExpenseMan` has been created, and I've shown the changes needed in each of `ExpenseMan's` views. You now have a fully working Newton application, one that provides multiple views of the data and allows for simple navigation through it. In the next article I add support for folders and the find mechanism to `ExpenseMan`. Until then, remember—"The pen is mightier than the mouse!" 🖋

Greg is cofounder of the ObjectPartnership, an object-oriented technology consulting firm in Litchfield, CT. While his native tongue is Smalltalk, he has recently been re-instantiated as a protoNewtonPioneer. As such, he responds to several messages, including `greg:DoDishes()`, `greg:PutWoodInStove()`, `greg:MakeCoffee()` and `greg:TurnOffPowerBook-WhileInBed()`. His handwriting style is mixed cursive & printed, his words are closely spaced and he recognizes handwriting slowly, more accurately.

The source code for ExpenseMan II can be found on the PIE Developers 2.2 source code disk.

CHUNKS

MORE AMAZING SOUP EXPERIMENTS!

George W. P. Henne

In the last issue of *PIE Developers*, I presented some soup benchmarks and found a few interesting ways to use them for best performance. A few questions were raised by the results that got me to run another series of tests to see if there was more to learn.

To start with, I set up a benchmark very similar to the one I ran last month. It creates, reads and writes entries to a 500-entry soup that's modeled after an inventory file. From the tests last month, I concluded that soups of this size exhibit similar performance characteristics to soups several times larger—soup speed is somewhat independent of the number of entries.

ARE SOUPS FASTER ON DIFFERENT ROM VERSIONS?

My first question is whether there is any difference in performance between the three most common versions of the Newton out there, 1.04, 1.05 and 1.11. I am fortunate enough to have access to Newtons with all three versions available, so I ran the tests on each.

In Table 1, rows two, six and seven show the results. In creating entries, 1.04 is the quickest. In the read and read/write tests, 1.05 is the fastest. In each case, the margin is slight—less than 10%. A difference this small is not noticeable by users and may be largely due to normal variations between runs.

The conclusion is that there is no significant difference in soup I/O speed between various ROM versions.

DO UNION SOUPS INCUR A PERFORMANCE PENALTY?

Another question is whether there is any performance loss in using union soups as opposed to dedicating the soup to a single store. Here tests two and four compare performance writing to a two MByte card, and tests three and five compare performance writing to the internal store.

In most of the tests, the non-union soup slightly outperforms the union soup, but not by enough to make a serious difference. There is only one case where the difference is significant: creating new soup entries on an internal store is faster if the soup is not a union soup. Once the entry is created, there are no further savings in reading or writing.

Overall, the good news seems to be there is no performance penalty in using union soups.

So far, this is turning out to be a pretty dull set of benchmarks. The answer to our questions always seems to be "it makes no difference." Let's look at some results that do.

Test#	Notes	Sequential		Random	Sequential		Random
		Create	Read		Read/Write	Read/Write	
1	1.05 union, 1 MB card	4.07	29.50	25.08			
2	1.11 union, 2 MB, read	2.21	25.55	22.29	5.91	5.28	
3	1.11 union, internal, read	4.10	25.73	21.98	7.34	6.31	
4	1.11 non-union, 2 MB read	2.35	26.52	23.35	5.90	5.20	
5	1.11 non-union, internal read	5.34	26.57	22.59	7.56	6.50	
6	1.05 union, 2 MB card read	2.15	28.87	25.38	5.98	5.50	
7	1.04 union, 2 MB card read	2.30	25.97	23.38	5.75	5.38	

Table 1 - Benchmark results (all times are in entries per second)

I next looked at the relative speeds of different stores in tests one, two and three. I ran my benchmark using the internal store and Apple's one and two MByte PCMCIA memory cards. For creating soup entries, the internal store and the one MByte card have a significant (almost two-fold) speed advantage over the two MByte card. Reading also had interesting results - the one MByte card has almost a 20% advantage over the other two.

The Apple one MByte SRAM card is much better if you are creating records. Furthermore, if you are reading data, the SRAM card is your quickest way to access it.

In most cases, developers won't have much choice in which store to use, but there are still a couple of useful hints here for design. First, it seems the type of store used is the limiting factor on soup access time. This indicates that NewtonScript isn't getting in the way too much (at least for reading) and that as faster stores are developed, the improvements will be immediately noticeable in applications that are I/O bound.

While the speed of reading entries from a soup is good, the speed of creating new entries and changing existing ones deserves comment. Writing a short new entry to a two MByte card takes about half a second. This amount can easily mount up to an unacceptably long response time for the user, especially if more than one entry is created.

Careful design certainly is needed here. Remembering the results of the last set of tests, it seems that where possible, it may be better to write a single large entry than a number of smaller ones.

I wonder if this doesn't uncover an area where the developers should go back and have another look. It shouldn't take five times as much time to write a record as it does to read it. The fact that there's only a 20% difference in a read and write operation to a two MByte card, compared to the internal store, tells me there's a lot of overhead somewhere that's not related to the actual physical write operation. With no overhead, I would expect the ratio to more closely mirror the actual speed of the stores, giving a ratio closer to two to one.

Overall, the conclusion is that soups work well under a variety of conditions. The Newton environment is designed to efficiently manage soups under different configurations. The consistent results that are obtained give me some confidence that future Newtons will continue to handle soups well. ☺

All numbers are in entries per second. Higher numbers are faster. The test soup is composed of 500 records. Each entry is in the form (Item, description & flags & price), where the description & flags & price field is 32 characters long. The size of the soup is 37,139 bytes, or about 75 bytes per entry. All tests were run with the Inspector turned off. Results are probably only significant to the whole number – fractional differences should be ignored. NTK 0.7b was used for developing the test applications.

The source code for George's tests can be found on the PIE Developers 2.2 source code disk.

Howard Oaklev

One of the most common complaints made by users about Newton applications, whether built-in or third party, is that they often appear to run very slowly. Five benchmarks are presented, in an effort to understand which basic NewtonScript operations are slowest, and comparisons are made with a bytecode interpreted language on the Macintosh.

Whatever misrepresentations can be supported by benchmarks, they can still be useful tools to examine performance aspects of different processors and their language systems. In this first look at the MessagePad, the tests are based on the Slopstone benchmark, a suite of low-level tests developed by Bruce Samuelson.

The NTK project and layout used are very simple. A single `protoApp` view holds two `protoStaticText` views (`bmDetails` and `bmresults`) to report the test and its result, and five `protoTextButtons` are used to actually run the benchmarks. Each button holds a single benchmark in its `ButtonClickScript`. These scripts all have the general structure:

```
func()
  begin local a, b, c, d, e, f;
    c := Ticks();
    // here goes the code to be tested
    d := Ticks();
    SetValue(bmtdetails, 'text, "name of test");
    e := (d - c)/60.0; f := NumberStr(e);
    SetValue(bmresults, 'text, f);
  end
```

The actual benchmark code blocks are:

```

for a := 1 to 16000 do // integer add with 238 1s
begin
  b := 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+
    1+1+1+1+1+1+1+1+1+1+1+1;
  b := b+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1;
    1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1;
  // 6 more identical lines
  b := b+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1;
end;

for a := 1 to 16000 do // floating add with 34 1.0s
begin
  b := 1.0+1.0+1.0+1.0+1.0+1.0+1.0+1.0+
    1.0+1.0+1.0+1.0+1.0+1.0+1.0+1.0;
  b := b+1.0+1.0+1.0+1.0+1.0+1.0+1.0+1.0+
    1.0+1.0+1.0+1.0+1.0+1.0+1.0+1.0;
  b := b+1.0+1.0+1.0+1.0+1.0+1.0;
end;

for a := 1 to 96000 do // string indexing
begin
  b := aStr[0];
  // 9 more identical lines
end;

for a := 1 to 16000 do // empty frame creation
begin
  b := { };
  // 19 more identical lines
end;

for a := 1 to 16000 do // cloning of empty frames
begin
  // g is created outside the loop
  b := Clone(g); // using g := { }
  // 9 more identical lines
end;

```


SPELUNKING WITH VIEWFRAME

Rob Bruce
CorNet
RobBruce@aol.com

In my current application I need to provide the user with several bits of functionality which already exist in the Newton. Rather than increasing my package size and wasting my time recreating these features with all new code, I decided to use built-in code instead. For instance, in our application we want to allow the user to phone a client. This is a simple task if you use the built-in Names application. However, if you have to write your own code it might take several hours to reproduce the call slip and all of its features.

The callslip function checks area codes against the current area code and modifies the telephone number accordingly. It also uses built-in system variables to store the telephone company access code and the user's calling card numbers. I wasn't interested in recreating all these capabilities. However, if I attempted to open callslip using Getroot().callslip:Open(), I would get an error -48200 - expecting a frame or array. After looking over callslip, I was worried that I was expected to pass it a Names soup entry. Since my record wasn't in the Names soup, I almost gave up. Fortunately, I remembered that you can write "call Jamie 555-1212" on the NotePad, tap Assist and the call slip comes up.

With that in mind, I started my Newton and tried once again to open the call slip using Getroot().callslip:Open(). When my Newton crashed, I looked at the package with ViewFrame to see what could be wrong. Several variables had been instantiated which weren't there before the package was launched. I assumed that one of them was the slot which callslip expected to be set before it opened.

Among the contenders was a slot called fields which seemed to have a lot of potential. I quickly wrote "call Jamie 555-1212" on the NotePad, tapped Assist, and then took another look with ViewFrame. With the call slip open, I looked at the frame which was set by the Assistant. It looked like this:

```
{alternatives: nil, dialdirect: "555-1212",  
  rawText: "Jamie", phone: "555-1212"}
```

Because it uses the Mac's keyboard for data entry, I entered my next experiment into the Inspector:

```
getroot().callslip.fields:={ alternatives: nil,  
  rawText: "Jamie", dialdirect: "717-555-1212"};  
getroot().callslip:Open();
```

A couple of quick tests determined that dialdirect was the number which the Inspector generated and that the phone number was an internal slot created by callslip. So much for that part of the problem.

When I tap the Call button, up pops a request for me to add this person to the Names soup. I wasn't really interested in having everyone see this dialog. I went back to ViewFrame, into the callslip._proto, looking for the viewChildren:

```
GetRoot().callslip._proto.viewChildren  
RO Array: viewChildren #8 Ref(2835473)  
0: (#2) callslip cancel button  
1: (#7) text: Call  
2: (#6) text: Options  
3: (#11)  
4: (#5) callFeedback  
5: (#14)  
6: (#7)  
7: (#8) text: Add to Names?
```

It looks like child #1 with the value text: Call is a good pick. It turns out to be a protoTextButton - a quick look at it's ButtonClickScript points me to a function called SpeedDial. Here's what SpeedDial looks like with ViewFrame:

```
GetRoot().callslip._proto.SpeedDial  
RO Frame: CodeBlock #5 Ref(2832353)  
func() begin  
  local speedyCallback;  
  IF NOT(userConfiguration.currentDialSpeaker = 'speaker)  
GOTO 33  
  SetValue(callFeedback, 'text,  
    "Will start dialing in 3 seconds");  
  RefreshViews();  
  Sleep(3 * 60);  
33: IF NOT(userConfiguration.useDialNavigator) GOTO 51  
  vars.navigators.dialNavigator  
GOTO 52  
52: IF NOT(nil) GOTO 71  
  :DialNavigate(callPhoneLine.entryLine.text);  
GOTO 87  
71: :Dial(callPhoneLine.entryLine.text,  
  userConfiguration.currentDialSpeaker);  
87: AddDeferredAction(AddCardHelper, [])  
end
```

I'm not interested in most of this code. However, the last line calls another function, AddCardHelper. As I ramble off to figure out what AddCardHelper does, I realize I don't really care - I just want it to disable it. In fact if I could have it close the call slip that would be great. So I add another line to my script to get:

```
getroot().callslip.fields:={  
  alternatives: nil, rawText: "Jamie",  
  dialdirect: "717-555-3333"};  
getroot().callslip:Open();  
getroot().callslip.AddCardHelper:=  
  func() begin callslip:close(); end;
```

This last line overrides the built-in AddCardHelper slot with my script.

It occurs to me that this is pretty messy. I've semi-permanently altered the function of this object by overriding the built-in slot. I need to remove my hack after I'm done but the timing must be right. The final product looks like this.

```
getroot().callslip.fields:={  
  alternatives: nil, rawText: "Jamie",  
  dialdirect: "717-555-3333"};  
getroot().callslip:Open();  
getroot().callslip.AddCardHelper:=  
  func() begin callslip:close();  
  addDelayedAction(func()  
    removeslot(getroot().callslip, 'AddCardHelper', [], 200);  
  end;
```

Now the patch disappears shortly after the call slip is dialed.

This short section of code saved me time and RAM, both valuable resources. My users don't have to worry about what "Add to Names?" means, and all that was added to my program was a short section of code rather than the complete logic which was already built into the Newton's ROM. ☺

Rob Bruce was recently hired by CorNet, a leading solutions provider in the Sales Automation market. He will be spearheading their Newton Development and striving to further the cause.

SUBJECT: STARTUPSCREEN NEWT

From: wkearney@access.digex.net (Bill Kearney)
Newsgroups: comp.sys.newton.programmer
Subject: Startupscreen Newt
Date: 31 Jan 1994 16:27:17 -0600

Here's a nifty little hack for the Inspector:

```
getroot().sleepscreen.viewsetupformscript:=  
func() begin  
    self.country:=userconfiguration.country;  
    userconfiguration.country:="Graceland";  
    inherited:=viewsetupformscript();  
end;
```

```
getroot().sleepscreen.viewsetuponescript:=  
func() begin  
    userconfiguration.country:=self.country;  
    inherited:=viewsetuponescript();  
end;
```

This lets you have the Newt startupscreen appear instead of the normal lightbulb logo. This is a temporary patch - resets kill it. I've put it into the install and remove scripts of an app of mine, and it works nicely. Remember to nil the slots on a remove. It's possible to replace the startupscreen with a totally different image, but I've not yet figured out how to do it without using around 30K of memory. The above patch appears to consume about 3 to 5K.

What it Does

The Newton has an app called `sleepscreen`. Guess what it does? It's the app that gets called when the switch on the side of the Newton is pressed while the Newton is asleep. The setup for `sleepscreen` looks at the `userconfiguration.country` slot and if it's `Graceland`, it swaps the normal startupscreen for the Newt. The above hack sets the existing country code aside in the base view of the `sleepscreen` app, in a slot named `country`. It then replaces the current value with `Graceland`. This lets another part of the `sleepscreen` app do it's thing. Once the app's done, it replaces the user's country code with the original value and exits.

Changing the screen to a different image entirely IS possible, it just takes up a lot of RAM. For that matter, changing the startup behavior of the Newton is possible as well. Stay tuned for a nifty little hack. 

How To DIM A TEXT BUTTON

Alb ric M ller
100015.304@compuserve.com

In one of the first Newton programs I wrote, I needed a button that could be dimmed under program control. I wished to show the user that the function associated with the button was unavailable, much like you can with the Macintosh System or with Windows. I studied the Newton Programmer's Guide and the nearest object I found was the "Text Button." I decided to modify it to create a new Dimmable Text Button object. This turned out to be very easy using NewtonScript's inheritance mechanism. All the slots that supplement or override the standard button object can be put in one frame that embodies the new object.

[The PIE user-interface police don't endorse the use of dimmed buttons. In fact, they don't like them. By the time you read this there may be some official user-interface guidelines that suggest how to indicate unavailable functions. Ed].

BUTTON STATE

In order for a Dimmable Text Button to work properly, the button must store its own state. This is done in a slot named `buttonActive` which is either `true` or `nil`. To access the button state we send the method `Enabled()` which simply returns the value of `buttonActive`:

```
Enabled:  
func()  
begin  
    return buttonActive;  
end
```

Note that the slot `buttonActive` is created dynamically while setting up the view. I do not define it in the template because it must be in RAM since it is modified by the methods `Enable()` and `Disable()`.

DIMMING THE BUTTON

I use a checkerboard-like mask to dim the text inside the button. This mask is generated on the fly by filling a suitable pattern checkerBoardPattern into a shape `buttonShape`. The slots are initialized in the `viewSetupFormScript`:

```
viewSetupFormScript:  
func()  
begin  
    // Create the button state slot  
    self.buttonActive := true;  
    // Create a rectangle from the button viewBounds  
    self.buttonShape := MakeRect(0, 0,  
        viewBounds.Right - viewBounds.Left,  
        viewBounds.Bottom - viewBounds.Top);  
    // Create a checkerboard pattern  
    self.checkerBoardPattern :=  
        SetClass(Clone("\uA55AA55AA55AA55"), 'pattern);  
    // Create viewFlags  
    self.viewFlags := Clone(viewFlags);  
end
```

The `buttonShape` slot is set to a rectangular shape defined by the `viewBounds` of the button. The trick to creating an eight by eight pixel pattern on the fly is to clone a string which is already a binary object and change its class to `'pattern'`. The string is specified using hexadecimal character codes whose binary representation is the pattern. Each two-digit hex code creates one byte (a row of eight pixels) of the pattern. Finally I set important `viewFlags` in the template and copy them in RAM: `vClippings` ensures that nothing is drawn outside the view and `vClickable` initially enables the button.

I dim the button text, depending on the value of `buttonActive`, in the `viewDrawScript`. This script is called every time the button needs to be redrawn:

```
viewDrawScript:  
func()  
begin  
    if NOT buttonActive  
    then :DrawShape(buttonShape,  
        {transferMode: modeBic,  
         fillPattern: checkerBoardPattern});  
end
```

`DrawShape` fills the rectangular `buttonShape` with the `checkerBoardPattern`. It clears only the black pixels within the button because I use the Bit-Clear transfer mode. This dims the text.

SETTING UP THE BUTTON STATE

The button is disabled or enabled by the following methods which modify its state:

```
Disable:
func()
begin
    buttonActive := nil;
    SetValue(self, 'viewFlags,
        Band(viewFlags, ~vClickable));
end
Enable:
func()
begin
    buttonActive := true;
    SetValue(self, 'viewFlags,
        Bor(viewFlags, vClickable));
end
```

I disable the button by resetting the `vClickable` flag so that the system ignores taps. I also enable the button by setting the `vClickable` flag. Note that it is important to use `SetValue()` because this dirties the view allowing it to be redrawn.

USING THE BUTTON

I have implemented the button as a user proto. After selecting it from the NTK prototype palette, place it in your own layout. NTK automatically creates a `_proto` slot that references the template. However you normally need to create three slots to override the inherited behavior:

- override `viewBounds` to place and size the button where you need it on your layout;
- override the `text` slot to set the button's name; and
- override the `buttonClickScript` slot which is needed to perform some useful actions.

For instance, you can modify the `buttonClickScript` to post an alert to check that the button is working correctly:

```
buttonClickScript:
func()
begin
    GetRoot():Notify(kNotifyQAlert,
        "DimButton", "Button works !");
end
```

As I already mentioned, send the `Enable()` and `Disable()` messages from appropriate code sections to control the button state, and `Enabled()` to query it.

CONCLUSION

This simple example shows the power of inheritance. We can reuse Apple's Text Button by overriding or adding only a few slots. The dimmable button itself can be reused as if it is one of NTK's objects. It would be handy if we could teach NTK to include the three required slots mentioned above automatically when placing the user-defined object in a layout. More generally, NTK should allow a programmer to define specific slot editors for user protos, so that we can handle them like system protos. ☞

Albéric Müller is a freelance consultant who specializes in data acquisition and database applications. He has worked with computers since the mid sixties, in several areas including power generation and aerospace.

The source code for Albéric's proto can be found on the PIE Developers 2.2 source code disk.

APPLICATION COOKBOOK

SOUPED-UP: A SOUP BROWSER

Theo Heselmans

Vers a Versa

vers.a.versa@applelink.apple.com

There are many tools dealing with soups: Souper, StewPot, Slurp, Pour, Ladle, and more. I needed a tool which would let me investigate my own soup entries one by one, and all slots at a time. That's why I started Souped-Up on a rainy Saturday afternoon, and posted it to AppleLink and CompuServe the next day. I got a pretty good response, a couple of bug reports, and requests for improvements. The version presented here fixes those bugs and adds some new features (although I have plenty of other new features in mind).

SOUP STRUCTURE

Every Newton *soup* has the same global structure (no different file formats to struggle with!). Each soup is stored on a *store*. Normally every soup, with the exception of the system soup, should exist on every available store. If a programmer follows Apple's directions as documented in the "Q&A Data Storage and Retrieval" their soup should always be on the Internal store and the Card store if there is a PCMCIA card.

Every soup can have one or more *indexes*, to speed up soup queries and sorts, and contains one or more *Entries*. An entry is really a record in a classic database environment, but each entry in a soup can have completely different content and structure. To access an entry you create a *cursor*. This is a context within which you can browse a soup. If you access a soup entry, a frame is created in RAM.

APPLICATION STRUCTURE

Souped-Up has the following parts:

- The project file.
- A main ProtoApp where the user can browse through the entries using the scroll arrows (see Figure 1).
- A linked layout to let the user select a store, a soup, an index and set some preferences (see Figure 2).
- A small linked layout for an 'About' protoFloatnGo.
- A Project Data file to set up some global symbols and a RemoveScript.
- A resource file with some icons and PICTs.

INTERFACE

When you first start Souped-Up, you need to select a store from a popup list. Then you select a soup from another popup list, and the index from a third list. The initial version of the program let the user select the soup from a popup. Since popups don't scroll, there was a problem – the user couldn't see all the soups if there was a very long list, especially on a card with backup data on it. So I split out the soup selection interface and put it on a separate layout, with the soup list in a `protoTextList`. Although this means a two-step method to get to your entry, I think it's more flexible, and it allows me to add more features without cluttering the main layout.

Another user interface problem is that an entry can be quite big (look at the `Userconfiguration` entry in the system soup). I need to be able to scroll, not only entry by entry, but inside an entry. It took me a while to figure out that in order to scroll a `clParagraphView` it has to be enclosed in a `clView`. It's the `clView` which is really scrolled. Using some `afterScripts` (a script which is executed at compile time), I added the up and down arrows to my scroll buttons:


```

AfterScript //for Up arrow
func() begin //found in NTK Definitions;
    thisview.icon:=ROM_uparrowbitmap
end

```

The scrolling itself is done by calculating the new viewOriginY of the clView named ListView:

```

buttonPressedScript: // of Up-arrow
func() begin
    // if not already scrolled to top
    if ListView.ViewOriginY>0 then begin
        //ScrollAmount set to 104 pixels
        local NewY:= ListView.ViewOriginY-
            ListView.ViewScrollAmount;
        if NewY < 0 then NewY:=0;
        // don't scroll beyond top
        ListView.SetOrigin(0,NewY); // set new Origin
        Refreshviews(); // Update immediately
    end; // (it's a buttonPressedScript)
end

```

Similar scripts are used for the down arrow.

The scroll arrows below the MessagePad screen are used to scroll from one entry to another, in a loop. To get this correct I first calculate the number of entries in a soup (TheSoupCount), and keep track of which entry is currently showing (prefsEntry.EntryNr). TheCursor contains the cursor I'm using. A similar method is used for the viewScrollUpScript:

```

viewScrollDownScript:
func() begin
    if TheSoupCount>0 then begin
        //if no records : no scroll needed
        if TheCursor.Next() then
            //if we're not at the last entry
            prefsEntry.EntryNr:=
                prefsEntry.EntryNr+1;
            //increment my current entry number
        else begin
            //if we are at the last entry
            TheCursor.Reset();
            //go back to the first entry
            prefsEntry.EntryNr:=1;
            //set current entry number back to 1
            GetRoot():Sysbeep();
            //let the user know we looped around
        end;
        :GetEntry(); //display new entry on screen
    end;
end

```

SELECTING A SOUP

I keep the latest settings in a preference entry in the system soup in the slot names prefsEntry. It contains a frame like this:

```

(tag:kPackageName, StoreNr:0, Soup:"system",
    IndexNr:0, EntryNr:1, LevelNr:1, ConvertDate:TRUE)

```

When you reopen Souped-Up, the same entry from the same soup in the same store is redisplayed. By storing the store number (zero for internal and one for card), the soup name and the index number, we have some flexibility, and the program is not restricted to a particular soup on a specific card. The trade-off is that because there is more checking to do, the code for retrieving the store, soup, index and entry is a little more complex. Remember that a particular soup or index could have

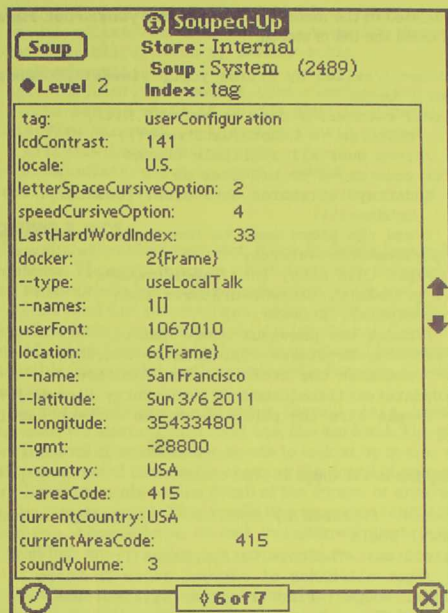


Figure 1 - Souped-Up main view

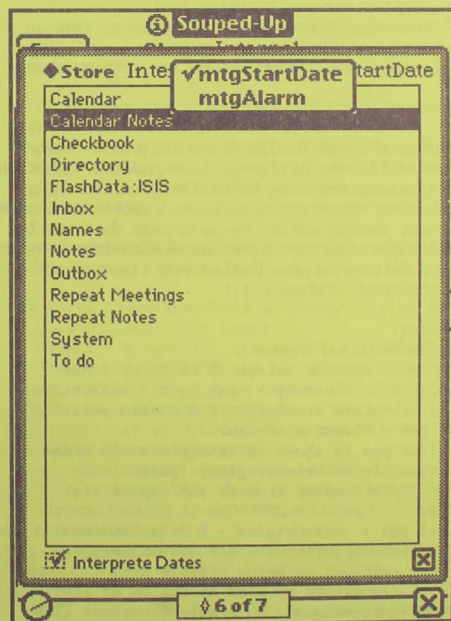


Figure 2 - Souped-Up linked subview

been deleted in the mean time, or the card yanked out. Here's how I build the list of stores:

```
GetStores://called by Stores popup viewSetupDoneScript
func() begin
    local counter, stores := array(0,Nil),
        nrStores := Length(GetStores())-1;
    //loop over all available stores
    for counter:=0 to nrStores do
        AddArraySlot(stores,GetStores()[counter]:
            GetName());
    //add the store name to the local stores array
    labelCommands:=stores;
    //put this array into labelCommands
    if prefsEntry.StoreNr>nrStores then
        prefsEntry.StoreNr:=0;
    //check the previous store number
    TheStore:=GetStores()[prefsEntry.StoreNr];
    //remember the Store in slot TheStore
    :UpdateText(labelCommands[prefsEntry.StoreNr]);
    //make sure the popup shows the correct label
end
```

Getting the list of soups is even easier:

```
GetSoups: //called by textlist's viewSetupDoneScript
func() begin
    listItems:=TheStore:GetSoupNames();
    //get all names of current store TheStore
    local SoupNr:=FindStringInArray(listItems,
        prefsEntry.Soup);
    //look up current soup name in array
    if SoupNr =nil then SoupNr:=0;
    //if not found, settle for first soup name
    TheSoup:=listItems[SoupNr];
    //remember the current soup name in TheSoup
    :UpdateList(SoupNr);
    //update the protoTextList and select correct
soup
end
```

Note the use of a protoTextList. Since this proto is poorly documented, I know a lot of people have problems with it. So did I at first, but experimenting the hard way bears its fruits.

This is my UpdateList function. It gets called when the listItems change, and has one parameter, the line to be selected. It also takes care of scrolling this line into view (note the use of the internal protoTextList slots lineHeight, selection and :SetupList):

```
UpdateList:
func(selectLine) begin
    //pass line to select (0 is first line)
    local h := lineHeight, pos := h * selectLine;
    //calculate lineHeight and wanted position
    if pos < viewOriginY then
        //if pos is above currently visible lines
        viewOriginY:=viewOriginY - pos;
        //position my line at the top of list
    else
        if pos > (viewOriginY + h * (viewLines-1)) then
            viewOriginY:=pos - h * (viewLines-1);
            //if pos is below visible area
            //position my line at bottom of list
        selection:=selectLine; //select correct line
        :SetupList(); //reconstruct viewbounds of list
        :RedoChildren();
        //install listItems by redrawing children
    end
```

Don't change the viewJustify to Horizontal Parent Full. Your lines don't get highlighted due to a bug in the protoTextList. Stay with Horizontal Parent Left.

Getting the list of indexes is similar to getting the soup list except for one little problem – TheStore:GetSoup(TheSoup):GetIndexes() returns an array of indexes. I need a separate function to get the name out of the array:

```
GetIndexString:
func(Index) begin
    //We pass one index from our array
    local path:=Index.path;
    //the path slot contains our index ref.
    if ClassOf(path)='PathExpr then
        Stringer(path) else sPrintObject(path);
    //this path can be a simple symbol
    //(then sPrintObject returns our string),
    //or a complete path expression
    //(then Stringer returns our string)
end
```

The only thing left to figure out is what to do when you select a store, soup or index – the buttonClickScript or the labelActionScript. When you select another store, a new soup list has to be build, and the first soup and index of the soup selected. When you select a different soup, only the index list has to be build and the first entry selected. The only thing to watch out for here is to set the settings for prefsEntry correctly.

SHOWING AN ENTRY

Once you know which soup you want to play with it's time for the hard part – interpreting the frame returned by the entry and showing it in a readable format in our clParagraphView. We start by setting up the cursor and informing the user on the main screen about the soup selection he/she made:

```
GetData:
func() begin
    local soup:=TheStore:GetSoup(TheSoup), Info;

    if prefsEntry.IndexNr then
        //if there is an index, we use it
        TheCursor:=query(soup,{type:'index,
            indexpath:soup:GetIndexes()
            [prefsEntry.IndexNr].path});
    else //if no index available,
        // use the default which is _uniqueID
        TheCursor:=query(soup,{type:'index});
    :CountSoup();
    //Count the number of entries and the total size
    Info:=GetStores()[prefsEntry.StoreNr]:GetName() &
        "\n" & TheSoup & " (" & TheSoupSize & ")\n"
        & (if prefsEntry.IndexNr then
            :GetIndexString(soup:GetIndexes()
                [prefsEntry.IndexNr])
            else
                "<None>");
    setValue(SoupInfo,'text,Info);
    //Assemble a nice string and show it
    if prefsEntry.EntryNr> TheSoupCount then
        prefsEntry.EntryNr:=1;
        //Make sure we have a valid Entry number
    if prefsEntry.EntryNr>1 then
        TheCursor:Move(prefsEntry.EntryNr-1);
        //Move to the correct entry number
    :GetEntry(); //finally show the entry
end
```


For those interested, counting a soup has to be done the hard way, using a loop. Since I have to loop anyway, I also accumulate the entry sizes:

```
CountSoup:
func () begin
    local count:=0, size:=0,
    entry, cursor:=TheCursor.Clone();
    //don't use clone(TheCursor)
    cursor.Reset(); //go to the first entry
    entry:=cursor.Entry(); //grab it
    while entry do begin //as long as there is one, go
        count := count + 1; //increment the counter
        size:=size+EntrySize(entry);
        //increase the total size
        entry:=cursor.Next() //fetch the next entry
    end;
    TheSoupCount:=count;
    //finally store the count and the size
    TheSoupSize:=size;
    //using locals in our loop increases speed !
end
```

The last step is creating the string to display a soup entry. This is done by looping over each slot in our entry. Note that if the information string gets big, we might have to stop somewhere because of insufficient memory. If someone knows a clean way to avoid these problems here, let the rest of us know!

```
GetEntry:
func() begin
    local slot,value;
    Append(nil,nil);
    //a small function to assemble or final string
    //by sending nil, we empty the string
    TheEntry:=TheCursor.Entry(); //get the entry frame
    if TheEntry = nil then
        //if there is no entry in this soup
        setvalue(Count,'Text','No Data');
        //show it at the bottom of the screen
    else begin
        IsDirty:=FrameDirty(TheEntry);
        //remember if the entry is dirty or not
        foreach slot,value in TheEntry do begin
            //loop over all slots
            :PrintOneObject(sPrintObject(slot),value,1);
            //investigate this slot !!
            //If we don't do this next part,
            //we get out of memory errors
            //and restart dialogs. If you have plenty
            //of frame heap, take this code out!
            //or change MaxChars (initially set to 1000 )
            if StrLen(TheData) >MaxChars then begin
                :Append("-----",UnicodeCR);
                //UnicodeCR is a Carriage Return character
                :Append("<More Slots here>",UnicodeCR);
                :Append("<Due to Memory problem,
                    have to stop here>",nil);
                Break;
            end; //here ends the dirty stuff
        end;
        if not IsDirty then EntryUndoChanges(TheEntry);
        //remove the frame the heap if not dirty
        setvalue(Count,'Text,Stringer(
            [(if IsDirty then " " else ""),
            prefsEntry.EntryNr," of ",TheSoupCount]));
        //show the user which entry number they see
        //add a prefix is the entry is dirty
    end;
```

```
ListView.ViewOriginY:=0; //scroll to top
setvalue(List,'text',TheData); //show the data
TheData:=nil; //clean up memory a bit
TheEntry:=nil;
setvalue(List,'viewbounds,List.viewbounds);
//to force recalc of bounds (bug in clpara)
ListView.ViewMaxY := List:LocalBox().bottom -
    ListView:LocalBox().bottom + 1;
//calculate the max. scroll limit
end
```

SHOWING A SLOT

As the script encounters each slot, it has to decide how to display its contents. The main function which deals with this problem is called `PrintOneObject`. It's a big function which just looks at the class and the primeclass of the object and adds an appropriate string to our total info slot (`TheData`). (I need to thank Matthew Dixon Cowles here, from whom I borrowed some ideas in his 'Pour' code). Let's examine this function a section at a time.

Besides the slot name and object, there is also a `level` parameter. We start off with level one, the top level. The user can select the level of detail he/she wants to look at by means of a small popup. Level two means showing the detail of a frame or array. Level three shows the detail of the frames or arrays within these frames or arrays. This smells like recursion (and it is). Inside `PrintOneObject`, we call `PrintOneObject` again, but one level higher.

```
PrintOneObject:
func(slot,object,level) begin

    if level > prefsEntry.LevelNr then return;
    //don't go beyond the requested level !
    if StrLen(TheData) > MaxChars then return;
    //don't exceed memory constraints (see GetData)
    local temp, slotname, value, prefix;
    local primeClass:=PrimClassOf(Object),
    class:=ClassOf(Object);
    //remember the primeClass and the Class of object
    prefix:=if slot = nil then ""
    else :Spaces(level) & slot & $: & UnicodeHT;
    //slotname is nil for arrays, so no prefix then
    //else the function Spaces gives us a nice prefix
    //including a tab character at the end
```

First we check for the simple classes: numbers, strings and symbols:

```
if primeClass='Immediate or class='String or
class='Symbol then begin
    temp:=if class='Int and object > 10000000 and
    object < 60000000 and prefsEntry.ConvertDate
    //check for a date (range 1923-2018)
    //and the user wants it converted
    //(this is a checkbox on the Select layout)
    then ShortDate(object) &&
    date(object).year else object;
    //convert to a date if wanted and possible
    :Append(prefix & :PrintObject(temp),if slot then
    UnicodeCR else ", ");
    //append string and add a return of needed
end;
```


Then we deal with arrays:

```
else if class = 'Array or primeClass = 'Array then
begin
:Append(prefix & length(object) & "[",nil);
//append the # of items in the array and a [
foreach value in object do
:PrintOneObject(nil,value,level+1);
//loop over items in the array and call self
:Append("]",UnicodeCR);//finally append final "]"
end;
```

Now it's up to the frames:

```
else if class = 'Frame or primeClass = 'Frame then
begin
:Append(prefix & length(object) &
"(Frame)",UnicodeCR);
//append # of items in the frame and (Frame)
foreach slotname, value in object do
:PrintOneObject(sPrintObject(slotname),
value,level+1);
//loop over items in frame and call self
end;
```

And finally we deal with all other classes:

```
else begin
temp:=:PrintObject(object);
//try to convert object to string directly
if strEqual(temp,"") then //if not possible
:Append(prefix & "<Unknown: " &
sPrintObject(primeClass)
& ", " & sPrintObject(class) &
">",if slot then UnicodeCR else ", ");
//append description of object
else
:Append(prefix & temp,if slot then
UnicodeCR else ", ");
//else append the string itself
end;
end
```

Let's finish with a small but necessary function.

sPrintObject() is an interesting built-in function but it does not return NIL or TRUE. That's why I created PrintObject(). The code speaks for itself:

```
PrintObject:
func(object) begin
if object = nil then return "NIL";
if object = true then "TRUE"
else sPrintObject(object);
end
```

CLEANING UP

We all know that each slot in our main view remains filled even when we close the application. I tend to have a slot called InitSlots, which sets all my extra data slots to nil. This function is called twice – once in my viewSetUpformScript to create those slots, and once in my viewQuitScript to clear them out. I also have a little RemoveScript in my Project Data to get rid of my entry in the System soup. The moment the user yanks out the card containing Souped-Up, or he/she removes Souped-Up, this script is executed:

```
RemoveScript:=
func(packageFrame) begin

local sysSoup :=
GetStores()[0]:GetSoup(ROM_SystemSoupName);
//Get the system soup
local cursor := Query(sysSoup,{type: 'index,
indexPath: 'tag, startKey: "Souped-Up:VaV"});
//Look for my entry
local prefsEntry := cursor:Entry();
//Get the Entry

if strEqual(prefsEntry.tag,"Souped-Up:VaV") then
//If it's mine, remove it from the soup!
EntryRemoveFromSoup(prefsEntry);
end;
```

CONCLUSION

I know Souped-Up can be improved a lot. One day you will see Souped-Up 2.0. My crystal ball (I really do have one!) tells me it could contain the following features:

- a soup menu with delete, duplicate and copy;
- an entry menu with delete, duplicate and edit;
- an overview of entries to pick from; and
- routing capability (and more).

This list is far from complete. Let me know if you have other requests. Let me know if you know better or faster ways to do things. A very important way to learn more about this exciting new machine is by sharing our knowledge. I'm just one of the happy few who had a chance to start a real-world project way back in August 1993. Work is beautiful when you're having fun at the same time, isn't it! 🍌

Theo Heselmans grew up with and Apple II and Mac, and now loves to play with this enormous Newton potential. Versa a Versa is a small company in Belgium, getting involved in Newton development. Their first vertical Newton application is already used in the field (for a water distribution company). Their second (for over 70 medical sales reps) will follow shortly.

The source code for Souped-Up 1.1 is on the PIE Developers 2.2 source code disk.

A very important way to learn more about this exciting new machine is by sharing our knowledge.

A DIFFERENT KIND OF CALCULATOR

Rob Lewis
R & D Business Systems
100140.1112@Compuserve.com

When I first started using the NTK, I decided that I needed to build a small application as a practice piece. Although it was not originally planned that way, this application has provided me with a test bed for learning nearly all aspects of Newton programming, including accepting written input, dragging views, persistent storage, the Intelligent Assistant, and providing help.

The application I chose to try out was a calculator. "Not another calculator!" I hear you groan. Read on - this one is different.

When I first got my Newton I was disappointed with the built-in calculator, not so much because of the lack of functions, but because it wasn't very Newton-like. What I wanted was a fairly minimalist floating box in which I could write out calculations and perform them. By doing away with the keypad and using written input, I figured it could be smaller. By writing into it I figured it would fit in with the Newton way of working better.

The first thing I realized was that any calculator that relied upon the user to write "+", "-", "*", or "/" on the screen was doomed, so I needed to provide those four basic functions using buttons. That gave me the basic interface structure of the calculator (see Figure 1):

- a draggable floating view;
- a text view for written entry and display; and
- four function buttons;

THE DRAGGABLE FLOATING VIEW

There is no built-in prototype template in the NTK for a draggable floating view. What you have to do is take a ProtoFloatNGo and add the code to make it draggable.

To build my ProtoDragNGo template, I created a new proto template by clicking on New Proto Template on the File menu and laying out a FloatNGo as its prototype. I then added a viewClickScript method which performs the drag. Note that because of a bug in the drag() function, you have to recreate the viewBounds after the drag is finished (this problem is documented in the NTK manual).

To prevent other clickable child views that you may place on this view from inheriting dragability, you also need to test that the pen tap is within the boundary area of the view. Notice that I seem to be checking to see if the pen tap is outside of the global

box. At first glance may seem to be redundant - if the user taps outside of the view, this method doesn't get called. In fact, GlobalBox returns coordinates of a box that is inside the rounded matte frame, but this method gets called if you tap in the frame. By checking to see if the tap is outside the global box, I am in effect checking to see if the tap is in the frame, which for a drag, is what I want.

```
ProtoDragNGo.viewClickScript :=  
func(unit)  
begin  
    local ParentBounds := :Parent():GlobalBox();  
    local X := GetPoint(firstX,unit);  
    local Y := GetPoint(firstY,unit);  
    local myGlobalBox := :GlobalBox();  
  
    if (X < myGlobalBox.left)  
    or (X > myGlobalBox.right)  
    or (Y < myGlobalBox.top)  
    or (Y > myGlobalBox.bottom) then  
        begin  
            :Drag(unit, ParentBounds);  
            myGlobalBox := :GlobalBox();  
            myGlobalBox.left :=  
                myGlobalBox.left - ParentBounds.left;  
            myGlobalBox.right :=  
                myGlobalBox.right - ParentBounds.left;  
            myGlobalBox.top :=  
                myGlobalBox.top - ParentBounds.top;  
            myGlobalBox.bottom :=  
                myGlobalBox.bottom - ParentBounds.top;  
            self.viewBounds := myGlobalBox;  
            true;  
        end;  
    end;  
end
```

I set the viewFlags as shown in Figure 2.

THE BASE VIEW

Once I define the user template, I am able to build the actual base view of the calculator by creating a new layout, clicking the User selection from the palette, and selecting ProtoDragNGo (see Figure 3 on the next page).

After creating a box which I can drag around and then close, I have to decide what functionality I want to add to it. I already know I want a data entry and display screen, and four function buttons, but in addition I want to add my company logo, and a help button.



Figure 1 - the calculator's main window

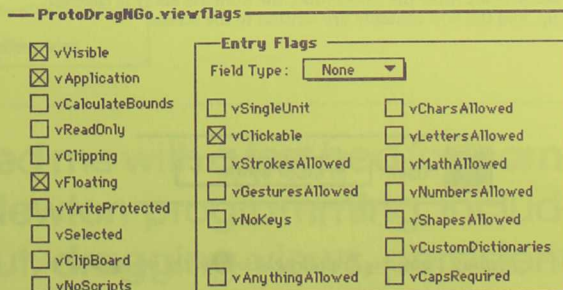


Figure 2 - flag settings for ProtoDragNGo

I use a `clParagraphView` for the data screen, `protoStaticText` for the logo (we have a very simple logo!), and `protoTextButtons` for the five buttons (see Figure 4). Note that you don't need to worry what size or shape to make the `ProtoDragNGo` template – you can drag it out to the dimensions you want just like the built-in prototypes. As you can see, I want a small application base view.

A `clParagraphView` can accept written text and display it. I want the user to be able to write a list of numbers in the view, but not write words. By setting the `viewFlags` to disable inputting words, the number recognition is dramatically improved. I also need to allow gestures so that text can be scrubbed or highlighted. These attributes are controlled by the flags for the `clParagraphView` (see Figure 5).

THE GLOBAL DATA SLOTS

I need to provide some slots to hold the information which is global to all of the views and functions in the application. I use a frame which contains both an array of strings to represent the numbers on the screen, a slot to hold the fixed point setting, and a slot which is an array containing the numbers on the screen in numerical format. These are declared in the `viewSetup-DoNeScript` of the base view (more about that script later):

```
calcFrame := {
  tag: kPackageName,
  strValues: ["0.00"],
  fix: 2
}
numValues := [0.00]
```

Note that the `calcFrame` also has a slot `tag`. This is used to identify the entry when the frame is copied into the soup for storage. The constant `kPackageName` is defined in the Project Data together with the symbol `kAppSymbol`.

DOING THE CALCULATIONS

The `DoCalc` function, which actually handles the calculations, is called by the `viewClickScript` of each of the calculator function buttons. They each pass a symbol parameter which represents their function. The first thing that `DoCalc` has to do is find out what is on the screen. It copies the text from the `clParagraphView CalcInput` and then calls a function which places the numbers it contains into the arrays `strValues` and `numValues`. It then tests to make sure that there are at least two numbers on the screen. If there are not, it simply returns, so the buttons do nothing.

I use a simple series of conditional statements to perform the correct calculation upon the last two numbers on the screen according to the button that was pressed. (The Sum calculation is performed via the Intelligent Assistant – I discuss this later in the article). After the calculation, the numeric array is cleaned up, and the new contents are written to the screen.

```
DoCalc :=
Func(op)
begin
  local NumStr := CalcInput.text;
  :getValues(NumStr);
  X := Length(numValues);
  If X > 1 then
  begin
    if op = 'Add then
      numValues[X-2] :=
        numValues[X-2] + numValues[X-1];
    else
      if op = 'Subtract then
        numValues[X-2] :=
          numValues[X-2] - numValues[X-1];
      else
        if op = 'Multiply then
          numValues[X-2] :=
            numValues[X-2] * numValues[X-1];
        else
          if op = 'Divide then
            numValues[X-2] :=
              numValues[X-2] / numValues[X-1];
          Y := 1; //remove one element
          if op = 'Sum then
          begin
            numValues[0] := :sumArray(numValues);
            Y := X - 1; //remove all but one element
            X := 2;
          end;
          ArrayRemoveCount(numValues, X-1, Y);
          :writescreen(calcFrame.fix);
        end;
      true;
    end
```

By creating a set of sub-functions, I manage to keep the main `DoCalc` function clean and readable. This is good programming practice – it allows us to concentrate on the task we want the function to perform without getting distracted by details. Of course, it is still no replacement for a good design.

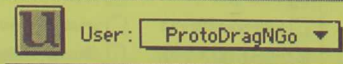


Figure 2 - selecting `ProtoDragNGo` from the user palette

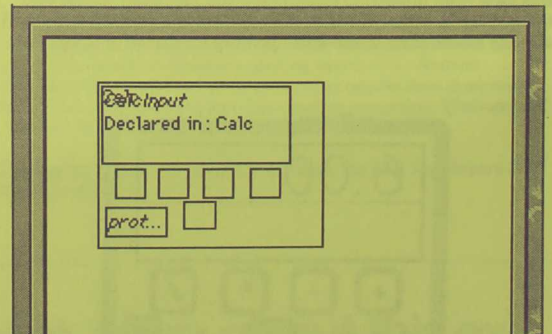


Figure 4 - `CalcInput clParagraphView` `viewFlags` settings

The `getValues` function copies the text from the screen into the two arrays:

```
getValues :=  
func(numStr)  
begin  
  numStr := :straightString(NumStr);  
  numStr := :fixNegatives(NumStr);  
  local index, value;  
  calcFrame.strValues := SplitString(NumStr);  
  setLength(numValues, length(calcFrame.strValues));  
  foreach index, value in calcFrame.strValues do  
    numValues[index] := StringToNumber(value);  
  end
```

The key to this function is `SplitString`, which takes a string and returns an array of all of the words in that string. I use it to place the numbers into the array in `calcFrame`, and then perform a `ForEach` to iterate through the array and copy each number into the `numValues` array. Unfortunately, life isn't quite so simple – I found that I have to do some tidying up of the string before `SplitString` parses properly.

There are two problems. One is that `SplitString` does not recognize newline as a word separator. The second is that when the Newton displays a negative number in the `paragraphview.text` slot, there is no leading space before the minus sign. This causes `SplitString` to treat that number and any previous number as one word.

Under normal use, with the user writing a series of numbers on the screen, there should be no newline characters in the string. However, if the user drags a column of figures into the `clParagraphView`, all we get are newlines. The first thing is to straighten out the line. `StraightString` simply uses `StrPos` to detect the presence of newlines in the string, replace each one with a space (the double ampersand concatenates with a space), and return the new string.

```
straightString :=  
func(myString)  
begin  
  local X;  
  while StrPos(myString, "\n", 1) do  
    begin  
      X := StrPos(myString, "\n", 0);  
      myString := Substr(myString, 0, X) &&  
        Substr(myString, X + 1, StrLen(myString) - X - 1);  
    end;  
  myString;  
end
```

The next job is to separate negative numbers. This function uses a different technique, since we are not removing the minus signs. It simply iterates through the string, inserting a space wherever it finds a negative.

```
fixNegatives := func(numStr)  
begin  
  for X := StrLen(numStr) - 1 to 1 by -1  
  do  
    if StrPos(numStr, "-", X) = X  
    then numStr := Substr(numStr, 0, X) &&  
      Substr(numStr, X, StrLen(numStr) - X);  
    numStr;  
  end
```

At the end, `DoCalcs` calls a function to write the result back to the screen. This function accepts the current fixed point setting as an argument. First, I have to make sure that `strValues` and `numValues` are the same length. I then use `foreach` to iterate through `numValues` copying the value into the same position in `strValues`. At the same time, I add the text of the number to the string `numStr` preparing to set the display.

CalcInput.viewFlags

- ☒ vVisible
- ☐ vApplication
- ☐ vCalculateBounds
- ☐ vReadOnly
- ☐ vClipping
- ☐ vFloating
- ☐ vWriteProtected
- ☐ vSelected
- ☒ vClipboard
- ☐ vNoScripts

Entry Flags	
Field Type:	None ▼
☒ vSingleUnit	☐ vCharsAllowed
☒ vClickable	☐ vLettersAllowed
☐ vStrokesAllowed	☒ vMathAllowed
☒ vGesturesAllowed	☒ vNumbersAllowed
☐ vNoKeys	☐ vShapesAllowed
☐ vAnythingAllowed	☐ vCustomDictionaries
	☐ vCapsRequired

Figure 5 - `clParagraphView` `viewFlags` settings

This application has provided me with a test bed for learning nearly all aspects of Newton programming, including accepting written input, dragging views, persistent storage, the Intelligent Assistant, and providing help.


```

writeScreen :=
func(myFix)
begin
    local NumStr := "";
    local index, value;
    setLength(calcFrame.strValues, length(numValues));
    foreach index, value in numValues do
    begin
        calcFrame.strValues[index] :=
            :convertNum(value, myFix);
        NumStr := NumStr && calcFrame.strValues[index];
    end;
    SetValue(CalcInput, 'text', NumStr);
end;

```

There is another function `convertNum` which takes as arguments a number and the fixed point setting, and returns the string representation of that number. This function uses `formattedNumberString` to extract the fixed point setting number. This setting is then used to create the format string for a second call to `formattedNumberString`, which converts the actual data value, which is returned. `SubStr` is used to remove the extra decimal place that I would otherwise get in the format string.

```

convertNum :=
func(number, fix)
begin
    local format := "%. ";
    local fixStr := formattedNumberStr(fix, "%.0f");
    format := format & substr(fixStr, 0, 1) & "f";
    formattedNumberStr(number, format);
end;

```

All I need to do now to make the calculator work is hook up the buttons. The `buttonClickScripts` of the buttons are similar to the Add button:

```

buttonClickScript :=
func()
begin
    :DoCalcs('Add');
end;

```

PERSISTENT STORAGE

Now I have a small calculator into which I can write numbers, press one of four keys to do calculations with the numbers, scrub the numbers with the pen, drag numbers into and out of the application, and drag the calculator around the screen. But that's just the easy part.

The first thing you notice is that every time the calculator is closed, it loses the numbers from the screen. I would like to save the data in a persistent store between uses. The amount of data that this application needs to save doesn't really necessitate a soup of its own, so I add one entry to the System soup. (If you have no packages installed on your card, you won't find a system soup - it only gets created to store an entry listing the applications on the card).

The System soup requires that each entry has a `tag` slot which identifies the owning application. The other slots are defined by the application. This means I can use `calcFrame`'s structure - it has a `tag` slot ready and waiting.

When the application is started, it must read the data from the soup. To avoid any unexpected behavior if, for example the application is on a card which is used in another Newton, I put a test in the `viewSetupDoneScript`, so that if the entry does not exist, a default one is created. On reflection, it might be appropriate

to put the entry into the soup on the default storage medium as part of the installation script. I may experiment with that in a later version and report back. In the meantime, I keep things simple and go straight for the internal System soup.

The `viewSetupDoneScript` is called after the base view and all its children are created, and all slots instantiated, but before it is displayed. Notice that I create the slots `sysSoup`, `cursor1`, `calcFrame` and `numValues` using the `self.` prefix. This ensures that the slots are global to the application so that other functions can reference them.

First I identify the internal system soup. The internal store is returned by `getStoreS()` [0]. The `sysTem` constant `ROM_systemSoupName` is guaranteed to return the name of the system soup (which is not necessarily guaranteed to stay the same, so always use this reference). I then create a cursor based upon a query to the System soup. This is an indexed query, which is based upon the `tag` slot in the soup. By specifying the `tag` slot, I can then move the cursor to the first record whose `tag` matches any value of my choosing. In this case, I look for the entry whose `tag` matches my application's `kPackageName` constant. If `gotoKey` succeeds in finding a matching tag, it puts the cursor at that entry. If it fails it puts the cursor at the first entry in the soup, so I have to check to make sure that this is my entry. (This is important if you are planning to delete the entry).

If the tag of the current entry does not match my `kPackageName`, I need to insert a new entry before I can continue. This is simply achieved using the `add()` function, which accepts as an argument the frame that is to be stored. Once the new frame is entered into the soup, I can then rerun the query so that the cursor is now positioned at the new entry. Having successfully added the new entry, I can continue to fetch the soup entry out of the soup into the `calcFrame`. All that remains is to copy the values from the array in `calcFrame` into the `numValues` array and then call `writeScreen`.

```

viewSetupDoneScript := func()
begin
    local index, value;
    self.myHelpBook := BuildContext(
        (_proto: GetRoot().TinyTim._proto,
         bookRef: theBook));
    self.sysSoup :=
        getStores()[0]:getSoup(ROM_systemSoupName);
    self.cursor1 := query(sysSoup,
        (type: 'index, indexpath: 'tag));
    cursor1.gotokey(kPackageName);
    if not strEqual(cursor1.entry().tag,
        kPackageName) then
    begin
        sysSoup.add((tag: kPackageName,
            strValues: ["0.00", fix: 2]));
        cursor1.reset(); cursor1.gotokey(kPackageName);
    end;
    self.calcFrame := cursor1.entry();
    self.numValues := [0.00];
    setLength(numValues, length(calcFrame.strValues));
    foreach index, value in calcFrame.strValues do
        numValues[index] := StringToNumber(value);
    :writeScreen(calcFrame.fix);
end

```

Saving the screen data back to the soup when the application closes is simple. Ignoring the `HelpBook` line for now, all I need to do is call `getValues` (in case the user has written some new data on the screen, but is closing without having done a calculation), and then inform the system that the entry copied to `calcFrame` is changed.

```
viewQuitScript :=
func()
begin
  myHelpBook:Close();
  :getValues(CalcInput.text);
  entryChange(calcFrame);
end
```

In an ideal world, the soup entry should be removed when the application is removed from the Newton. Unfortunately, at present the `RemoveScript` gets called when the card that the application is on is removed from the Newton. There are good reasons for this – the Newton has to know that it cannot call the application from the Intelligent Assistant, for example, but it means that the `RemoveScript` is not a good place to delete data that is supposed to be persistent.

Most Newton applications either leave the data in the soup, and rely on the user to delete the data if they really want to, or they include a procedure to remove the soup data. Since the amount of data that I put in the soup is small (compared to the size of the code required to add another button or Intelligent Assistant feature to remove it), I use the former method. I do provide a warning about this and offer to assist any users that have problems.

INTELLIGENT ASSISTANT

In addition to the four basic functions of the calculator, I also provide two other features. I want the user to be able to alter the fixed point setting (the code is in place, but the user at present has no way of reaching it) and I want to add the ability to sum all of the values in the screen. Since I want to keep the calculator as small as possible, I don't want to add any more buttons. This leaves me with two choices: either I can build another view which can pop up with the extra buttons on it, or I can use the Intelligent Assistant. I think that the Intelligent Assistant is by far the best way of adding these extras to an application.

There is still a lot about the Intelligent Assistant which is undocumented. I am not claiming that I know it all, or that my implementation is optimum. As Ford Prefect once said, "Don't knock it, it works!", and if it saves others the head scratching I had to go through just to get this far, then it is worthwhile describing it.

Keywords and Action Templates

The first thing to do is to decide what key words you want Assist to respond to. To set the decimal places, I use "fix" and to add up the values on the screen, I use "tot". In working on this part of the program, I came across an interesting issue – applications competing for the use of the same keywords. Since I already have `MobileMath` from `MobileSoft Corp.` (and quite like its features), I have to make sure that I don't pick any keywords that `MobileMath` uses, which include the obvious "sum". This suggests that Apple should provide a registration for keywords in the same way as they register signatures.

Having picked my keywords, I can create my action templates. For setting preferences, such as decimal places, in the calculator, I use:

```
calcSet := (
  value: "Calc setup action",
  isa: 'dyna_user_action',
  lexicon: ["fix"]
)
```

and for performing calculations I use:

```
calcAction := (
  value: "Calc action",
  isa: 'dyna_user_action',
  lexicon: ["tot"]
)
```

I also need to describe the fixed point arguments that I allow the user to enter:

```
fixNum := (
  value: "Fixed places",
  isa: 'dyna_user_obj',
  lexicon: ["0", "1", "2", "3", "4", "5"]
);
```

These are entered as slots in the base view.

Task Templates and the InstallScript

I also need to provide a taskTemplate for each of the actions. These are registered with the system as part of the `InstallScript`. The `InstallScript` goes into the Project Data file and accepts (from the system) a frame called `PackageFrame` which describes the package that is being installed. Note that the base view in the application is referenced as `packageFrame.theForm` – this is a handy way of referring to the slots in the application.

```
PackageFrame := (
  app: Calc:R&D,
  text: "Calc",
  icon: ...,
  theForm: {the base view frame},
  autoclose: noAutoClose,
  devinstallscript:
  Installscript:
  packageID:
  packageType: 3,
  formcontext: NIL)
```

The `InstallScript` is listed on the next page. The primary `act` slot in the taskTemplate frame refers to the action templates that I include in the application. The `preconditions` slot lists the phrases which must be present for Assist to consider this template.

In the fixed point template I specify an `'action` phrase and a `'fixNumber` phrase. The action phrase I expect is the one in the lexicon of the `calcSet` frame in the base view, so the corresponding item in the signature slot refers to that frame. The `'fixNumber` phrase I expect is one of the numbers in the lexicon of the `fixNum` frame, so that goes in the corresponding signature slot.

When I first got my Newton I was disappointed with the built-in calculator, not so much because of the lack of functions, but because it wasn't very Newton-like.

The postparse slot refers to a function in the base view which gets called when Assist decides that his is the task that the user means. appSym is set to the Application symbol kAppSymbol. Note the extra comma after that slot entry – it needs to be there.

```
InstallScript := func(packageFrame)
begin
  packageFrame.taskTemplateID := regTaskTemplate((
    value: "Calc fixed point",
    isa: 'task_template',
    primary_act: packageFrame.theForm.calcSet,
    preconditions: ['action', 'fixNumber'],
    signature: [packageFrame.theForm.calcSet,
      'packageFrame.theForm.fixNum],
    postparse: packageFrame.theForm.Handlefixset,
    appSym: packageFrame.(kAppSymbol),
  ));
  packageFrame.taskTemplateID2 := regTaskTemplate((
    value: "Calc sum",
    isa: 'task_template',
    primary_act: packageFrame.theForm.calcAction,
    preconditions: ['action'],
    signature: [packageFrame.theForm.calcAction],
    postparse: packageFrame.theForm.HandleSumAction,
    appSym: packageFrame.(kAppSymbol),
  ));
end;
```

The RemoveScript

The RemoveScript simply unregisters the two task templates if they are registered. They are identified by the two taskTemplateID slots in the packageFrame that are created by the InstallScript. This script is also placed in the Project Data file.

```
RemoveScript := func(packageFrame)
begin
  // IA
  if packageFrame.taskTemplateID then
    UnRegTaskTemplate(packageFrame.taskTemplateID);
  if packageFrame.taskTemplateID2 then
    UnRegTaskTemplate(packageFrame.taskTemplateID2);
  end;
```

Now that all the preliminaries are taken care of, I need to actually create the code that is run once the Intelligent Assist uses the above information to select my application out of the hat. First I need to define the postparse methods that I declare in the taskTemplates. These are entered as slots in the base view of the application, but are run in the context of the taskTemplate, not the application, so be careful.

The Fix Command

For handling the "fix" command, I define handleFixSet:

```
handleFixSet :=
func()
begin
  getRoot().(kAppSymbol):fixSet(self);
  getRoot().assistant:Close();
end
```

This in turn calls the function fixSet in the application, passing the taskFrame (self, the taskFrame, is the current context) as the argument. FixSet is also defined in the application's base view. This is where the actual work gets done. FixSet receives the taskTemplate as its argument. It looks something like this:

```
{ value: "Calc fixed point",
  isa: task_template,
  primary_act: (value: "Calc Setup action",
    isa: dyna_user_action,
    Lexicon: ["fix"]),
  preconditions: [action],
  signature: [(value: "Calc Setup action",
    isa: dyna_user_action,
    Lexicon: ["fix"])],
  postParse: <CodeBlock, 0 args #4411601>,
  appSym: NIL,
  parse: [(value: "Calc Setup action",
    isa: dyna_user_action,
    Lexicon: [#4411581])],
  input: [(value: "Calc Setup action",
    isa: dyna_user_action,
    Lexicon: ["fix"]),
    (isa: (isa: (#3CF92D)),
    value: "Fixed places",
    Lexicon: ["0", "1", "2", "3", "4", "5"])],
  raw: [(value: "Calc Setup action",
    isa: dyna_user_action,
    Lexicon: ["fix"]),
    (lex:
      (isa: (isa: (isa: (#3CE365))),
      value: "2"))],
  phrases: ["fix", "2"],
  noisewords: [],
  OrigPhrase: ["fix", "2"],
  action: [(value: "Calc Setup action",
    isa: dyna_user_action,
    Lexicon: [#4411581])]
```

The two arrays that I find most useful are input and phrases. What I do is iterate through the input array using forEach, and present each template in the array to the system function isa to test if it is an action or if it matches one of my "Fixed places" settings. The actual values that I need to work with are in the corresponding array slots in phrases.

Once I have an action, and that action is "fix", and I have a number, I update calcFrame and rewrite the screen.

```
fixSet :=
func(taskTemplate)
begin
  local template, myAction, index, value;
  local myFix := calcFrame.myFix;
  foreach index, template in taskTemplate.input do
    begin
      if isa(template, 'action') then
        myAction := taskTemplate.phrases[index];
      else
        if strEqual(template.value, "Fixed places") then
          myFix :=
            stringToNumber(taskTemplate.phrases[index]);
        else
          getRoot():confirm("Bad Fix",
            "Fixed point setting out of range", self, 'badFix');
        end;
      if strEqual(myAction, "fix") then
        begin
          calcFrame.fix := myFix;
          :writeScreen(myFix);
        end;
      end
```

The confirm method provides a popup warning to inform the user that I don't like what they wrote. It isn't ideal (it has OK and Cancel buttons) but it is reasonably effective for the few times it is likely to be used.

The Sum Command

The sum action is largely the same, with a postparse function handleSumAction which is called in the context of the action template by Assist, which in turn calls the sumAction function in the application's context.

```
handleSumAction :=
func()
begin
  getRoot().(kAppSymbol):sumAction(self);
  getRoot().assistant:Close();
end

sumAction :=
func(taskTemplate)
begin
  local template, myAction, myValue;
  if Length(taskTemplate.input) = 1 then
    begin
      template := taskTemplate.Input[0];
      if isa(template, 'action') then
        begin
          myAction := taskTemplate.phrases[0];
          if strEqual(myAction, "tot") then
            :doCalcs('Sum');
          end;
        end;
      end;
    end
  end
```

In sumAction I check that only one phrase is passed. If that phrase is the action "tot" then I call DoCalcs, passing the symbol 'Sum.

That's all it takes to add Intelligent Assist to my application. If the user writes "fix 3" and taps Assist, the decimal precision on the calculator changes to three. If the user has a series of numbers on the calculator screen and writes "tot" and taps Assist, the calculator totals the numbers on the screen. These phrases should be written in the NotePad or the Assist popup, not in the calculator screen.

HELP

How does the user know what to do? To help them figure it out I implemented a help function. The first thing to do is create the help book source file. I experimented a bit before I found a comfortable combination of tools for this. I have Claris Works (for the Claris XTND system that Book Maker requires), but Book Maker cannot accept Claris input (you tell me...). Book Maker, with the Claris XTND system installed, accepts Word for Windows input (although only Word 2, not 6, and any PICTs I embed in my Word 6 file get lost by the time they reach the Mac).

I've found that the most convenient method for help files is to create the document in Claris Works and then Save As MacWrite. Book Maker goes *much* faster with MacWrite input

than with Word for Windows. As a last resort, you can create a file with TeachText (even without XTND), but that means no bold headings and no pictures. Any pictures you include must be in PICT format. That's all clear isn't it?

To produce the few simple pages of help that this application requires is quite simple.

- Bookmaker requires .title and .isbn commands, although help books don't need them, so put what you like as their arguments.
- Prefix each help page with .subject 1 StartsPage followed by the page title (which I set in bold), and then .story followed by the actual text.
- To add a picture, use .picture and paste the picture on the following line. It's possible to reference a PICT file, but watch out - bookmaker isn't very good with path names.
- I wanted to give my help screen the personal touch, so I signed my name on the Newton, backed up the note with the connection kit, and copied the signature into the help file using the Clipboard.
- Save the file in a suitable format, such as MacWrite, and drop its icon on Book Maker, after turning off all other applications (unless you have lots of memory).
- Set the size to Help under Options, and click Do It.
- Open the output file with TeachText and copy the output into your Project Data file via the clipboard.
- Then just above the help section in the Project Data file type the line:

```
refNum := OpenResFileX(HOME & "CalcHelpText.f");
```

where CalcHelpText.f is the Book Maker output file. The NTK compiler needs this to find the PICT resource for the picture.

- You also need to include the helpbook references in the viewSetupDoneScript and the viewQuitScript, and add a slot to the base view initialized like this:

```
theBook := Book
```

- Finally, add an Icon (I won't bore you with that since Howard Oakley covered it in *PIE Developers* 2.1).

There you have it. A complete application with Intelligent Assist, Help and persistent memory.

ACKNOWLEDGEMENTS

I owe thanks to the members of the NEWTDEV forum on CompuServe who put up with a barrage of questions (and kindly provided some useful answers) while I was going through this learning curve. 🐉

Rob Lewis was the technical architect of Digital Equipment Corporation's European Wholesale Data Warehousing program. He has given all that up to start R & D Business Systems, a company specializing in Business Productivity applications for Newton and MS-Windows. Guess which he finds more interesting.

To produce the few simple pages of help that this application requires is quite simple.

THOUGHTS ON INTERFACE: WAITING FOR NEWTON

by Howard Oakley

One important topic which my previous article ("Thoughts On The Human Interface", *PIE Developers* 2.1) did not consider is what to do when the user has to wait for your application to do something. Whether your code is crunching through calculations, or communicating with another device, it is essential that any substantial delay is properly indicated by your program's interface. That is the focus of this article.

THE DUMB WAITER

The dumb approach to a wait is simply to do nothing special, and to allow the user to wonder how much longer he or she has to wait, or whether your application has crashed. A simple and purposeless example of what can happen is in a `proto-TextButton` with the following `buttonClickScript`:

```
func()
begin
  local a := 1;
  local b := 1;
  local d;
  for c := 1 to 2500 do
    begin
      // a modestly time-consuming task
    end;
  end
```

When the user taps the button, it becomes highlighted, and remains so until the method finishes. If you can be certain that this takes only a second or two at most, there is no need for feedback to inform the user as to what he or she should be doing. If the script takes any longer, some users may despair and reset their Newtons.

THE MINI WAITER

If you wish to inform the user that he or she has to wait, you can display a small floating window containing an icon or other indicator. This is probably best used in situations where the wait is fairly short. You can opt for any one of a range of different icons or messages.

My preference is for a simple "universal" icon (shown below). Adding a window of this sort to a `buttonClickScript` makes it look something like this:

```
func()
begin
  local a := 1;
  local b := 1;
  local d;
  minifloat:Open(); // open the floating window
  for c := 1 to 2500 do
    begin
      // a fairly time-consuming task
    end;
  minifloat:Close(); // and close it again
end
```



Since nothing changes in the floating window, it can be implemented as a prototype, a simple `protoFloater` containing a `clPictureView` with the requisite icon inside it:

```
_userproto000 :=
(
  viewBounds: (left: 10, top: 10,
               right: 50, bottom: 50),
  viewClickScript:
  func(unit)
  begin
    self:Drag(unit, nil);
    self:Dirty();
  end,
  viewflags: 2640,
  _proto: protoFloater
);

_view000 := /* child of _userproto000 */
(
  viewflags: 1,
  icon: GetPictAsBits("eggtime3", nil),
  viewFormat: 256,
  viewBounds: (left: 4, top: 4, right: 36, bottom: 36),
  viewclass: 76
);
```

THE ANIMATED WAITER

For longer waits of indeterminate length, a static floating window has two major disadvantages – it fails to reassure the user that the application is still working, and the time spent waiting seems much longer than it is. On the other hand, we do not know how long the whole process will take, so we cannot use any form of progress dialog. In a desktop system, we would probably put up a simple alert, saying that the application is busy and ask the user to wait, and then animate the cursor (or a similar device) to establish that the application is still processing (and possibly ease the user's boredom).

Since the Newton does not have a global cursor, the answer again appears to be to display a small floating window, and to animate an icon (in fact, as a picture) within that small window. Performance considerations need to be kept in mind – whatever we do should not swamp the screen or use unnecessary amounts of processor time or battery power.

My suggested solution is to use the same floater as in the mini waiter, and to run the picture through a looped-back series of icons shown below. The code to do this in our `buttonClickScript` might be:

```
func()
begin
  local a := 1;
  local b := 1;
  local c, d, e;
  anifloat:Open(); // open the floating window
  for e := 1 to 10 do
    begin
      anifloat:waitUpdateScript();
      // call it to update periodically
      for c := 1 to 250 do
        begin
          // a very time-consuming task
        end;
      end;
      anifloat:Close(); // and close it again
    end
  end
```



(It is also possible to install an idle event to call the icon animation routine `waitUpdateScript()`, but my attempts to do this from within a floating window proved unsuccessful).

With this alternative, the waiting window is best implemented as a layout (template) rather than as a prototype:

```
anifloat := /* child of waitertest */
(viewBounds: {top: 40, left: 10,
  right: 50, bottom: 80},
viewClickScript:
  func(unit)
    begin
      self:Drag(unit, nil);
      self:Dirty();
    end,
viewFlags: 2640,
waitUpdateScript:
  func()
    begin
      anipict:SwapIcon(); // this calls the icon animation
    end,
  _proto: protoFloater,
  debug: "anifloat"
);

anipict := /* child of anifloat */
(viewFlags: 1,
icon: GetPictAsBits("eggtime1", nil),
viewFormat: 256,
viewBounds: {top: 4, left: 4, right: 36, bottom: 36},
iconIndex: 1,
eggtime1: GetPictAsBits("eggtime1", nil),
  // these are the icons shown
eggtime2: GetPictAsBits("eggtime2", nil),
eggtime3: GetPictAsBits("eggtime3", nil),
eggtime4: GetPictAsBits("eggtime4", nil),
eggtime5: GetPictAsBits("eggtime5", nil),
SwapIcon:
  func()
    begin
      local a := (self.iconIndex mod 5) + 1;
      local b := NumberStr(a);
      local c := "self.icon := eggtime" & b & ";";
      local ans;
      SetValue(self, 'iconIndex, a);
      self.temp := compile(c);
      ans := self:temp();
      self:Dirty(); // since we have not used SetValue
      RefreshViews(); // these two calls are vital
    end,
viewClass: 76,
debug: "anipict"
);
```

Note the ingenuity required to programmatically swap the icon being displayed in `SwapIcon()`. This uses on-the-fly compilation of a constructed string to set the new icon, which appears to be the simplest approach (my attempts to construct and access an array of icons were frustrated).

THE PROGRESSIVE WAITER

The final interface tool which we need, to be able to handle other common waiting scenarios, is a full-blown progress dialog. This can be used when we can divide the waiting period into a number of discrete stages or steps. It is an exact parallel of progress dialogs on desktop systems.

My proposed design for a generic progress dialog is shown in the right-hand column. It has three elements inside a floating window:

- the same egg-timer icon used in the other waiters (for consistency and instant recognition);
- a text message which can be changed as the application desires; and
- an unadorned `protoGauge` view to register progress.

Once again, economy in processor time and battery power are important – I refrain from including icon animation since it only duplicates the feedback already provided by the gauge and text.

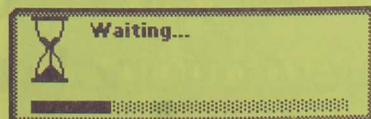
You might call this dialog using the following `buttonClickScript`:

```
func()
  begin
    local a := 1;
    local b := 1;
    local c, d, e, f;
    protofloat:Open(); // open the floating window
    protofloat:progressMax(10);
    // set its maximum progress value = all done
    for f := 1 to 10 do
      begin
        e := NumberStr(f);
        e := "Running loop number" & e;
        protofloat:changeWaitText(e, (f - 1));
        // change the text and progress value
        for c := 1 to 250 do
          begin
            // a very time-consuming discrete task

          end;
        end;
      end;
    protofloat:Close(); // and close it again
  end,
```

The progress dialog is slightly more complex:

```
protofloat := /* child of waitertest */
(viewBounds: {top: 62, left: -4, right: 168, bottom: 114},
viewClickScript:
  func(unit)
    begin
      self:Drag(unit, nil);
      self:Dirty();
    end,
viewFlags: 2640,
changeWaitText:
  func(words, level)
    begin
      local a := (level * 100) div self.maxGauge;
      SetValue(protext, 'text, words); // set text content
      SetValue(protoGauge, 'viewValue, a); // set gauge value
      RefreshViews(); // required to update the window
    end,
```




```

progressMax:
func(maximum)
begin
    SetValue(self, 'maxGauge', maximum);
end,
maxGauge: 100,
_proto: protoFloater,
debug: "profloat"
);

proicon := /* child of profloat */
{viewFlags: 1,
 icon: GetPictAsBits("eggtime3", nil),
 viewFormat: 256,
 viewBounds: (top: 4, left: 4, right: 36, bottom: 36),
 viewclass: 76,
 debug: "proicon"
};
// View proicon is accessible from profloat

protext := /* child of profloat */
{text: "Waiting...",
 viewBounds: (top: 4, left: 40, right: 176, bottom: 36),
 _proto: protoStaticText,
 debug: "protext"
};
// View protext is accessible from profloat

progauge := /* child of profloat */
{
    viewSetupFormScript:
    func()
    begin
    end,
    viewValue: 0,
    viewBounds: (top: 42, left: 8, right: 164, bottom: 54),
    maxValue: 100,
    _proto: protoGauge,
    debug: "progauge"
};
// View progauge is accessible from profloat

```

MAKING MESSAGES CLEAR

Although I expect others to improve upon these first efforts at waiting indicators, there are some design principles which I would commend to those tempted to do so.

Icons

Selecting good icons is very difficult. For this particular article, a large number of possibilities came to mind, and colleagues in the Newton Developers' Forum on CompuServe suggested many

others. Icons need to be clear, distinguishable, as international as possible, consistent and concise. I chose the egg-timer because it comes closest to meeting those requirements. In particular, clock-like icons, which would have been my first choice, are quite inappropriate on a Newton – they are easily confused with the other clock icons already used. Elaborating ideas of sleeping Newtons, the letter "Z", and so on, all failed on a number of counts.

Parsimony

As with all human interface elements, it is easy to go overboard (and have animations of leaping Newtons and other graphic paraphernalia). On desktop computers, these can be great fun. But Newton users may not thank you for imposing on them something which prolongs their waiting time by 50% and which eats away at their precious battery power. I think it's essential to choose interface designs which conserve processor cycles and battery life by minimizing the screen area used for changing graphics and also the code required.

You can check this claim by running the application included on the *PIE Developers 2.2* source code disk and timing the period required for each of the demonstrations. Even the icon animation and progress dialogs only add about 12% to the running time of the routines. Before you try this, I would recommend that you put your stopwatch away, estimate how long each wait seems, and compare your guesses with the actual times taken.

Consistency

The best human interface design is also highly consistent, both between and within applications. An excellent example of this is the comparison between the Macintosh's filing dialogs, and the anarchy which existed in the early days of the Amiga – the absence of any standard filing dialogs significantly impaired the Amiga's human interface, and diminished its air of professionalism. I believe the time has come to incorporate waiting dialogs into the Newton ROM, at least at the three levels shown here, to enhance its interface. Before that happens, you would do well to use a system such as that described here.

CONCLUSION

I present a three-level system for implementing waiting indicators in the Newton human interface. Each is suitable for different circumstances, ranging from a wait of a few seconds to a few minutes, which might be necessary when undertaking a complex connection with an external device. Each can be incorporated into your own code very simply and then called with ease. I welcome your improvements. 🐉

Howard's samples can be found on the PIE Developers 2.2 source code disk.

Selecting good icons is very difficult. Icons need to be clear, distinguishable, as international as possible, consistent and concise.

TIPS AND TECHNIQUES

by Howard Oakley

SAVING PREFERENCES

As long as your application has a reasonably small amount of preferences data to save, it's best to save it in the System soup. Make sure that you have a tag slot with your package name, adding the data from a slot called, say, prefs:

```
( tag: kPackageName,  
  firstData: theData,  
  values: [...] // and so on for the data  
)
```

A method like the following reads your preferences or creates new ones according to a default set:

```
func()  
begin  
  //get our preference data from the SystemSoup  
  //create if necessary  
  local sysSoup :=  
    GetStores()[0]:GetSoup(ROM_SystemSoupName);  
  local cursor :=  
    Query(sysSoup, {type: 'index', indexPath: 'tag',  
      startKey: kAppString,  
      endTest: func(e) NOT StrEqual(e.tag, kAppString)});  
  prefsEntry := cursor:Entry();  
  if NOT prefsEntry then  
    prefsEntry := sysSoup:Add(  
      {tag: kAppString, startWithSound: nil});  
end
```

Of Benchmarks and Bottlenecks

Continued from Page 25

It is easy to dismiss these results, saying that bytecode interpreters are always inefficient. However, comparison between STA and other Mac languages illustrates that this is not necessarily so. STA takes 11 seconds to run the add float benchmark, while Macintosh Common Lisp (which compiles to a dynamic code run in native mode) takes around 10 seconds, and compiled MPW Pascal ranges from 7 to 10 seconds, depending on the length of the floating point variables. STA at least manages to achieve impressive efficiency.

CONCLUSION

Running low-level benchmarks in NewtonScript shows that many common operations, such as the addition of integer and floating point numbers, are very slow relative to a similar bytecode-interpreted language running on a CISC chip. While this illustrates that computationally intensive applications are unlikely to be immediately successful, this may improve in a future native mode compiler. 🐉

EVERY BYTE COUNTS

If you want to give your users every possible byte of memory, you can close the Extras drawer (saving something like 4K) once your application is open by calling

```
GetRoot().ExtrasDrawer:Close();
```

BOOK MAKER 1.0b7 LIMITS

The version of Book Maker which ships with NTK 1.0b7 can run into problems when stories become larger than 32K. Try inserting additional .story lines to break the text up. This is a bug; a fix is promised in a future release.

BOOK MAKER MARGINS

The recommended margin setting for Book Maker is 3.33 inches. Note that this is the width inside the margins, not the width of the margins.

UNIQUE ICONS FOR BOOKS

To assign a unique icon to a book, add the PICT resource file to the book project as if it is an application (adding a mask PICT if you wish), then add the following to the book preamble:

```
book.icon := GetPICTAsBits("PictName", maskToo);
```

where maskToo is a Boolean indicating the availability of a mask. If this is passed as nil, then a mask is constructed.

Remember when you first got your
Newton™, how you had a million
great ideas for applications?
But after conquering NTK, how
would you distribute them?

Now there is a way...

NewtNet

Global Internet Distribution

For a copy of our FAQ for Developers:
internet ftp: [nuchat.sccsi.com](ftp://nuchat.sccsi.com), /pub
internet mail: newtnet@sccsi.com
phone: (713) 661-3301
fax: (713) 661-0633

Let The Proliferation Begin...

Newton™ is a trademark of Apple Computer, Inc.

LISP and Symbolic Computation: An International Journal

Editors-in-Chief : **Robert R. Kessler**, *University of Utah*
and **Carolyn Talcott**, *Stanford University*

LISP and Symbolic Computation is an international journal that presents a forum for current and evolving symbolic computing, focusing on LISP and object-oriented programming.

Key subject areas and topics covered in the journal include:

- Programming language notations for symbolic computing, such as data abstraction, parallelism, lazy evaluation, infinite data objects, self reference, message passing, generic functions, inheritance, encapsulation, protection, meta-objects.
- Implementations and techniques, such as specialized architectures, compiler design, combinatory models, garbage collection, storage management, performance analysis, smalltalks, flavors, common loops, etc.

- Programming logics, such as semantics and reasoning about programs, types and type inference.
- Programming environments and tools, such as knowledge-based programming tools, program transformation, specifications, debugging tools.
- Applications and experience with symbolic computing, such as real time programming, artificial intelligence tools, experience with LISP, object-oriented programming, window systems, user interfaces, operating systems, parallel/distributed computing.

LISP and Symbolic Computation is abstracted/indexed in: *Engineering Index*; *INSPEC Information Services*; *Artificial Intelligence Abstracts*; *Computer and Information Systems Abstracts*; *COMPENDEX Plus Database*; *ACM Guide to Computing Literature*; and *ACM Computing Reviews*.

Subscription Information

Volume 7, 1994 (4 issues)
ISSN 0892-4635

Institutional Rate: \$211.00/Dfl. 404.00
Individual Rate: \$100.00/Dfl. 235.00

*Subscription prices include
shipping and handling*

**CONTACT THE PUBLISHER
FOR A
FREE SAMPLE COPY**

Submissions

Authors should submit three (3) copies
as indicated:

Two (2) hard copies to:

Mrs. Karen Cullen
LISP AND SYMBOLIC COMPUTATION
Kluwer Academic Publishers
101 Philip Drive
Norwell, MA 02061, USA

One (1) electronic copy to:

Carolyn Talcott
clt@sail.stanford.edu

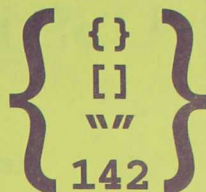


Kluwer Academic Publishers

North America: Journal Department • P.O. Box 358, Accord Station • Hingham, MA 02018-0358 •
phone: (617) 871-6600 • fax (617) 871-6528 • e-mail: kluwer@world.std.com
Rest of World: P.O. Box 322 • 3300 AH Dordrecht, The Netherlands • phone: (31) 78 524400 •
fax: (31) 78 183273

VIEWFRAME™

Newton™ Debugging and Exploration Tool



"I've been using Jason Harper's ViewFrame pkg for 2 or 3 weeks and I don't leave home without it! ViewFrame has saved me days of development!
Two thumbs up, Jason!" [unrehearsed and unsolicited]
Bill Judd, Cypress Research Corporation

Print listing
Fax
>Inspector
Inspector print()
Sort by slot
Sort by value

◆ View as: viewJustify

22
0x16
= vjCenterH
+ vjCenterV
+ vjParentCenterH

Size: 13 x 13

Mask:

Hit:

Hit:

Hit:

Entry in Internal: System

216 bytes (17 text)

tag: "UserDictionary"

data: <dictdata: #159>

_uniqueID: 3

_modTime: 47260508

```
func(unit) begin
  IF NOT(TrackHitLite(unit)) GOTO 16
  buttonClickScript();
  hitLite(nil);
  16: true
end
```

```
GetRoot()
GetGlobals()
GetStores()
GetUnionSoup("")
GetView("viewFrontMost")
debug("")
/RemoveSlot("%")
/DV("%")
/ViewView("%")
```

ViewFrame is a set of three independent Newton-resident programs for exploring and debugging Newton software. ViewFrame lets you browse a Newton's complete object space, forwards and backwards, examining and modifying almost all objects in that space in multiple formats. You can see more things, more clearly and easily than you can with the Inspector, without typing up the serial port.

ViewFrame Editor 0.5

```
for i := 0 to 369 do begin
  local x := compile($@ & i);
  x := call x with ();
  if classof(x) = 'frame and
  x.sendFrameType exists
  then begin
    playsoundsync(x);
    sleep(30);
  end;
end
```

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

The ViewFrame Editor lets you enter complete Newton programs, run them, debug them, and send the results to ViewFrame for further examination.

The Programmer's Keyboard has a variety of shortcuts for entering NewtonScript phrases and keywords, to simplify entering and editing programs.

ViewFrame is ideal for examining and debugging NewtonScript programs, especially when the Newton's serial port is unavailable for connection to the Inspector. ViewFrame and the Programmer's Keyboard can be accessed from other programs using a well-documented slot-level interface. Developers can also define custom object viewers and object modifications using a second slot-level interface.

ViewFrame is \$70. The price for PIE Developers subscribers is \$60 plus \$5 shipping and handling. We accept US bank-drawn checks and major credit cards only. Site licenses are available. Look for the ViewFrame demo on AppleLink, AOL, and CompuServe or contact us.

Creative Digital Systems

All good stuff, no fluff.

293 Corbett Avenue
San Francisco, CA 94114

cdssem@aol.com • cds.sem@applelink.apple.com

415.621.4252

415.621.4922 (fax)

cds@netcom.com • 72722.3225@compuserve.com